UNIVERSITY OF OSLO
Department of Informatics

Open-source virtualization

Functionality and performance of Qemu/KVM, Xen, Libvirt and VirtualBox

Master Thesis

Jan Magnus Granberg Opsahl

Spring 2013

Abstract

The main purpose of this thesis is to evaluate the most common open-source virtualization technologies available. Namely Qemu/KVM, Xen, Libvirt and VirtualBox. The thesis investigates the various virtualization platforms in terms of architecture and overall usage. Before further investigating the platforms through a series of benchmarks.

The results gathered from the benchmarks presents Qemu/KVM as the better in terms of performance in most of the benchmarks. Of these we can count the CPU- and memory intensive benchmarks. For the file-systems benchmarks, Xen delivers performance that is above the other examined virtualization platforms. The results also highlight the performance gained with processor additions such as Intel Extended Page Tables and AMD Rapid Virtualization Indexing, to enable hardware assisted paging.

Acknowledgments

First and foremost, I thank my thesis supervisor Knut Omang, for his insights, directing me in the right direction when I have lost my way, and most importantly for being incredibly patient.

I would also like to thank my fellow students at the Dmms laboratory for a thriving environment, inspiring discussions and their feedback.

Last I thank my family for their patience, understanding and endless support during my thesis work. Most importantly I thank my wonderful girlfriend Ingebjørg Miljeteig for believing in me and her enduring support and love.

May 2. 2013.
Jan Magnus Granberg Opsahl

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Since the advent of hardware extensions to the X86 processor architecture to enable hardware supported virtualization, virtualization has had an immense growth on X86 based computer architectures. In particular with the development of Kernel-based Virtual Machine (KVM) for the Linux operating system, as well as the increased interest in cloud computing. The benefits of using virtualization technology are typically considered to be server consolidation, isolation and ease of management. Allowing users to have concurrent operating systems on one computer, have potentially hazardous applications run in a sandbox, all of which can be managed from a single terminal.

This thesis will look further into the background for virtualization and why it is useful. I will present a detailed view of the most popular open-source virtualization suites, Xen, KVM, Libvirt and VirtualBox. All of which will be compared to each other with regard to their architecture and usage. The main part of this thesis will be the performance measurement and benchmarks performed on the aforementioned virtualization platforms. These benchmarks will be performed using popular performance measurement tools such as High Performance Linpack (HPL), LMBench and IOZone.

Previous work that has measured the performance of these virtualization suites have presented results that shows that Xen performs the best. With the rapid development in both virtualization platforms and hardware extensions to the X86 architecture that has occurred since the previous work was conducted. All of these virtualization platforms has taken full use of hardware extensions that allow virtual machines to maintain their own page-tables, giving rise to performance increases. For that reason it is suspected that performance among these virtualization platform have changed.

From the results of the benchmarks conducted in this thesis there is a clear indication of KVM having surpassed Xen in performance, in CPU usage and memory utilization. File system benchmarks indicate more ambiguous results that favor both virtualization platforms. In terms of usage, the development of Libvirt and Virtual Machine Manager has made both Xen and KVM more available for a wider audience that want to utilize virtualization platforms.

1.2 Motivation

The motivation for performing this work is twofold. First we want to present the various virtualization platforms to see what differentiates them from each other. Which of the hypervisors are the most intrusive on the host operating system, and what are the key architectural traits of the virtualization platforms. We also want to see if they stand up to each other when compared in terms of usage, which one is the most usable for system administrators that are not familiar with a Linux terminal, and administrators that want to use virtualization technology to its fullest. As well we investigate the various features of the platforms, which supports live migration, snapshots, and PCI passthrough, through a basic comparison.

The second, and most important of the motivational factors for this thesis is the performance of the various virtualization platforms. How do these virtualization platforms compare to each other in various aspects of performance. With all of the platforms having their own architectural traits that require different approaches to virtualization, with regard to processor sensitive instructions. How does the number of processor cores affect performance of the guest. Do the various disk configurations available for virtualization platforms affect the performance of disk and file systems operations. In addition to other constraints that might be imposed by the various tools that utilize the same virtualization platform, i.e. Qemu versus Libvirt configuration of guest machines.

With many enhancements that has been developed for the virtualization platforms and hardware. It is suspected that performance has changed drastically from when previous work was conducted. Newer benchmarks will either confirm previous benchmarks or present new findings that indicate where open-source virtualization technology stands with regard to performance. It will also be possible to indicate if any of the virtualization platforms are better suited for various workloads, i.e. CPU intensive or disk intensive workloads.

1.3 Previous work

There has been a lot of work on measuring the performance of virtual machines, of which many focus on the performance with regard to high-performance computing (HPC), as a basis for cloud computing technologies, live migration, and Xen and KVM performance and comparison. The work in this thesis does build upon some of the previous work that has been done.

Deshane et al.[13] compared Xen and KVM to each other with focus on performance and scalability. Che et al.[10] compared Xen and KVM to each other and measured performance of both. Xu et al.[67] measured Xen and KVM as well in addition to VMWare. Che et al.[11] measured the performance of Xen, KVM and OpenVZ in 2010 with focus on three different approaches to virtualization. Tafa et al.[47] compared Xen and KVM with both full- and paravirtualization in addition to OpenVZ to each other and evaluated CPU and memory performance under FTP and HTTP workloads.

In [15, 39, 68] the authors have studied the various available virtualization platforms for usage in HPC. While in [27, 7, 9] the authors have focused on presenting the available tools for managing cloud computing platforms that utilize virtualization platforms such as Xen and KVM, among them is Libvirt.

1.4 Thesis structure

Following this section the thesis will be structured as follows:

Chapter 2 will present some background for virtualization. The requirements for virtualization to be successful, what it is and the various types of virtualization. In addition to the history of virtualization from the 1960s and up. And a closing look at various benefits and some of the most popular virtualization platforms.

Chapter 3 will have an in depth look at the Qemu/KVM, Xen, Libvirt and VirtualBox virtualization platforms. The platforms will be examined in terms of their architecture and usage, and ultimately how they compare to each other on the these two points.

Chapter 4 will feature a more thorough presentation of related and previous work, before looking into the design of the benchmarks. How the various virtualization suites will be benchmarked, which benchmarks will be used and finally how the measurements will be performed.

Chapter 5 presents the results from the benchmarks, with comments to the results.

Chapter 6 concludes the thesis with a conclusion with regard to the compared virtualization platforms and the benchmarks.

Appendix features additional results and tables with numerical results for some of the benchmarks. In addition to some installation instructions for the platforms used.

Chapter 2

Background

2.1 Introduction

This chapter will look into what virtualization is. It will establish a vocabulary and talk about common terms and definitions when dealing with virtualization technology, as well as the theory behind virtualization and what is required of a computer architecture to support virtualization. We will then cover the history of virtualization from the 1960s and up, to why virtualization has been a hot topic in the IBM mainframe community and with the advent of hardware assisted virtualization for the X86 architecture, why it has become so popular once again. Lastly we will look at the various types of virtualization, the advantages and disadvantages, and the different solutions that exist.

2.2 Terms and definitions

Firstly I want to establish some vocabulary and clarify a some terms that will be used in this thesis, that could spark some confusion to the reader.

What I want to clarify are the three terms *virtual machine*, *virtual machine monitor* and *hypervisor*.

- **Virtual Machine (VM)** A virtual machine is the machine that is being run itself. It is a machine that is "fooled"[42] into thinking that it is being run on real hardware, when in fact the machine is running its software or operating system on an abstraction layer that sits between the VM and the hardware.
- **Virtual Machine Monitor (VMM)**[1] The VMM is what sits between the VM and the hardware. There are two types of VMMs that we differentiate among;[17]
  - *Native* sits directly on top of the hardware. Mostly used in traditional virtualization systems from the 1960s from IBM and the modern virtualization suite Xen.
  - *Hosted* sits on top of an existing operating system. The most prominent in modern virtualization systems.

[1] Not to be confused with virtual memory manager.

The abbreviation VMM can be both *virtual machine manager* and *virtual machine monitor*, they are both the same. Historically the term Control Program (CP) was also used to describe a VMM.[12]

- **Hypervisor** This is the same as a VMM. The term was first used in the 60s[66], and is today sometimes used to describe virtualization solutions such as the Xen hypervisor.

2.2.1 On Intel VT and AMD-V

Throughout this thesis I am going to mention Intel VT and AMD-V quite often. So to clarify some confusion that might arise when the reader inevitably is going to read about VT-x at some point and perhaps AMD SVM at some other point.

Firstly, Intel VT and the differences here. The reader will most likely stumble upon the terms Intel VT-x, VT-i, VT-d and VT-c at some point. This paper will almost exclusively deal with VT-x. VT-x is the technology from Intel that represents their virtualization solution for the x86 platform. VT-i is a similar to VT-x, except that it is the virtualization technology for the Intel Itanium processor architecture. VT-d is Intel's virtualization technology for directed I/O, which deals with the I/O memory management unit (IOMMU). VT-C is Intel's virtualization technology for connectivity, and is used for I/O virtualization and networks.

The virtualization technology from AMD is known as AMD-V. However, AMD firstly called their virtualization technology "Pacifica" and published their technology as AMD SVM (Secure Virtual Machine), before it became AMD-V. Some documentation for the AMD virtualization suite still refers to the AMD virtualization technology as "Pacifica" and SVM. For all further purposes in this thesis AMD-V will be used. Like Intel, AMD has also made technology for the IOMMU, which is known as AMD-Vi (notice the small 'i').

2.3 What is virtualization?

When asked this question regarding my thesis the default reply has more than often become, the technology that allows for one computer to simultaneously exist inside another.

Virtualization is a software technique that has been around for almost half a century now, that allows for the creation of, one or more, virtual machines that exist inside one computer. It was first developed to take better use of available hardware, which was often costly and often subject to stringent scheduling. Which in turn meant that developers often would have to wait several days for a computer to become available for them to test and run their programs, often leading to less than optimal usage of the computer. In addition to allow several users to have their own terminal, and as a consequence have multiple users of a single computer.

2.3.1 Characteristics

Virtualization has its roots in the concept of virtual memory and time sharing systems. In the early days of computing real memory was expensive, and a solution which would let a program larger than the available memory to be run

was strongly needed. The solution was to develop virtual memory and paging techniques that would make it easy to have large programs in memory and to enable multiprogramming. Another technology which helped virtualization forward was *time sharing*, both time sharing and virtual memory concepts will be covered later in this paper.

In an article from 1974 by Gerald J. Popek and Robert P. Goldberg[36] a model for a virtual machine and machines which can be virtualized is presented. They give three characteristics for a VMM:

- **Equivalence** This characteristic means that any program that is being run under the VMM should exhibit behavior that is identical to the behavior that program would give, were it run on the original machine. However, this behavior is not necessarily identical when there are other VMs present in the VMM that might cause scheduling conflicts between present VMs.

- **Efficiency** The VMM must be able to run a statistically dominant subset of instructions directly on the real processor, without any software intervention by the VMM.

- **Resource Control** The VMM should be in complete control of the system resources, meaning that it should not be possible for any running program to access resources that was not explicitly allocated to it. And the VMM should be, under certain circumstances, able to regain control of already allocated resources.

2.3.2 Virtualization Theorems

For a machine to be effectively virtualized, Popek and Goldberg came forth with three theorems which in turn is based on three classifications:

- *Privileged instructions:* Instructions that trap if and only if the state of the processor S_1 is in supervisor mode and S_2 is in user mode.

- *Control sensitive instructions:* Instructions that tries to change or affect the processor mode without going through the trapping sequence.

- *Behavior sensitive instructions:* Instructions that depends upon the configuration of resources in the system.

The theorems which can be derived from these classifications follows:

- **Theorem 1** *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

  This theorem states that to build a sufficient VMM all sensitive instructions should always trap and pass on control to the VMM, non-privileged instructions should be handled natively. This also gives rise to the *trap-and-emulate* technique in virtualization.

- **Theorem 2** *A conventional third generation computer is recursively virtualizable if it is: (a) virtualizable, and (b) a VMM without any timing dependencies can be constructed for it*

  This theorem presents the requirements for recursive virtualization, in which a VMM is itself run under another VMM. As long as the three characteristics of a virtual machine holds true, a recursive VMM can be constructed. The number of nested VMMs is dependent upon the amount of available memory.

- **Theorem 3** *A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions*

  Presents the requirements for a hybrid virtual machine (HVM) to be constructed. Here all instructions are interpreted, rather than being run natively, all sensitive instructions are trapped and simulated. As done in paravirtualization techniques.

All of these theorems and classifications presented by Popek and Goldberg, can be used to deduce whether a machine is virtualizable or not. The X86 platform did not meet these requirements and could not be virtualized in the classical sense of *trap-and-emulate*.

2.3.3 Types of VMMs

A VMM is often classified as a Type I, Type II or Hybrid VMM. These types were defined in Robert P. Goldberg's thesis in 1973[17], and are defined as follows.

- **Type I VMM** Runs directly on the machine, meaning that the VMM has has direct communication with the hardware. The OS/Kernel must perform scheduling and resource allocation for all VMs.

- **Type II VMM** Runs as an application inside the host OS. All resource allocation and scheduling facilities are offered by the host OS. Additionally all requirements for a Type I VMM must be met for a Type II VMM to be supported.

- **HVM**[2] Is usually implemented when neither a Type I or Type II VMM can be supported by the processor. All privileged instructions are interpreted in software, and special drivers have to written for the operating system running as a guest.

Those that are familiar with certain virtualization tools, which will be covered later in this chapter, might already have connected the types to the virtualization tools they are familiar with. Examples of a Type 1 VMM are, Xen, VMware ESX Server and virtualization solutions offered by IBM such as z/VM. Examples of Type II VMMs are VMWare workstation, and VirtualBox and KVM, both rely on kernel modules and a user application. And lastly an example of a HVM is solution is, Xen, using paravirtualized drivers.

[2]Hybrid Virtual Machine

2.3.4 Types of virtualization

This section will sum up the various types of virtualization that exist. It will also give a minor introduction to some of the various terms that will be used to describe various VMs and virtualization techniques.

Hardware virtualization

Hardware virtualization is the "classic" type of virtualization, it hides the underlying machine from guest operating systems or VMs, by displaying an abstract machine to the VM. It is also known as platform virtualization.

This type of virtualization was the first type of virtualization that was developed when virtualization technology was explored and developed in the 1960s and 1970s. Nowadays this type of virtualization technology is still the most prominent in use and under development. With the advent of hardware assisted virtualization, this type of virtualization has come back into the spotlight in the mid 2000s.

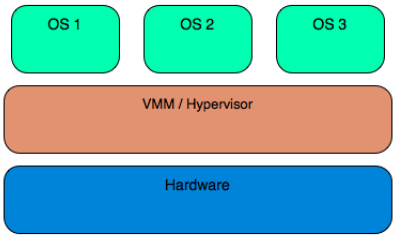

Figure 2.1: The typical architecture of virtualization software. Hardware at the bottom and an abstract layer to expose a VM, which runs its own operating system on what it thinks is real hardware.

We can differentiate between a few different types of hardware virtualization, hardware-assisted virtualization, full virtualization, paravirtualization, operating system level virtualization and partial virtualization.

Hardware-assisted virtualization Hardware assisted virtualization utilizes facilities available in the hardware to distinguish between guest and host mode on the processor. This makes it possible to construct VMMs that use the classic *trap-and-emulate* technique.

This is the type of virtualization that was used on the virtualization systems of the 1960s and onward. While the X86 processor did not have such facilities available in its original design, recent hardware extensions has made virtualization possible on the X86 architecture.

Full virtualization Full virtualization is a type of virtualization that allows operating systems, and its kernel to run unmodified in a VM. The VMM presents an interface to the VM that is indistinguishable from physical hardware. In the case of the X86 architecture, virtualization is only possible if either using the hardware extensions or either techniques such as paravirtualization or binary translation.

Paravirtualization Paravirtualization does not virtualize a complete computer environment in the manner that full virtualization does. Instead it provides a software interface, or API, that is similar to the underlying hardware. The guest knows that it is being virtualized, which means that the guest OS will need some modification to be able to execute. Through the interface the guest can make direct calls to the underlying hardware. If the virtualization layer supports direct communication with the hardware through available facilities, e.g. hardware-assisted virtualization, the calls can be mapped directly to the hardware.

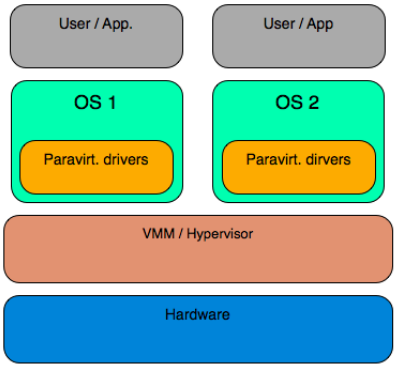

Figure 2.2: Paravirtualization abstraction showing the modified drivers that need be present in the OS.

Operating system-level virtualization OS virtualization isolates processes within a virtual execution environment by monitoring their interaction with the underlying OS. It is a technique which allows for several isolated instances run in user-space, often known as containers and jails. Examples are the UNIX `chroot` command, FreeBSD jails, Linux Containers (LXC), OpenVZ and Solaris Containers. Similarly to hardware virtualization traits, the applications should exhibit an behavior that is same as if it were to be run on an unvirtualized system[25].

OS virtualization can be classified by approaches of two dimensions; host-independence and completeness. Host-independence provides a private virtual namespace for the resources that are referenced by the application. Completeness virtualizes all the OS resources for the application.

Partial virtualization Using partial virtualization only a subset of the OS resources are virtualized. One of the key examples being virtual memory where each process has its own private memory space. Other techniques of partial virtualization may be used to tighten the security of system, by restricting the scope of certain processes.

Software-based Virtualization

Application virtualization Application virtualization is not as much of a virtualization technology, but more of a wrapper term for some applications and technologies. These technologies are used to improve upon the compatibility

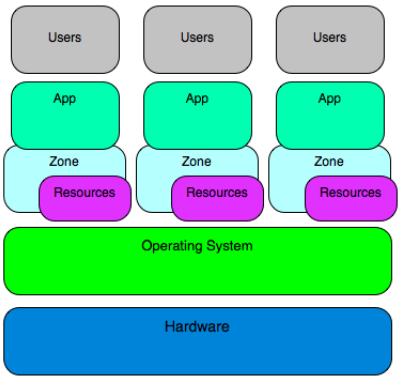

Figure 2.3: Operating system level virtualization.

and portability of applications. They are encapsulated from the OS, with a "virtualization" layer that is used when the application is executed, thus the application executes as if it were installed. This layer replaces the run-time environment that ordinarily is installed along with the application, or already present on the system. The Wine application suite for Linux is an example of such an application, others are the portable version of the Mozilla Firefox browser and Texmaker Latex editing software[49, 30].

Desktop virtualization

Desktop virtualization separates the logical desktop from the physical machine, using the well known client-server model of computing. The client may be a thin-terminal or another terminal of some kind, or a piece of software which enables the client to communicate with the required computer. VMs are mostly used to give users what they perceive as their own private computer system, when it is running on a virtualization server. This is generally the model which is used in cloud computing environment, independently of the cloud computing model.

One of the most prominent types of desktop virtualization is Virtual Desktop Infrastructure (VDI)[29] which enables the hosting of a desktop OS inside a VM that is run on a remote server. This makes it easy to migrate workloads between computers and data centers, and resources are shared easily among users on the same server.

2.4 Background for virtualization

2.4.1 Historic background for virtualization

In the late 1950s and early 1960s, computers were as large as cabins. Miles away from the personal computers, laptops, netbooks, and tablets of today. CPU time was often sparse and expensive, and only one person, or one task, could use a computer at the time. Programs were written specifically to one machine, and programmers often had to be present during execution of their programs in case anything did not go as planned.

The need for a system which would allow programmers to interact directly with a computer led to the development of time sharing systems, most notably the Compatible Time Sharing System (CTSS). This would allow several jobs and users to be present at the same time during execution, to also make better use of otherwise expensive hardware. The need also existed to have facilities to develop and research operating systems[38]. Giving each user a completely isolated system were erroneous programs could be developed and debugged, without affecting the other users present on the system.

This resulted in the development of virtual machine systems. In the case of the IBM System 360 family of virtual machine products, the control program (CP) or what now would be considered a hypervisor, would allow for several fully virtualized systems to coexist on the same S/360. Each virtual machine had its own memory range and its own storage space. At the same time each of the virtual machines were capable of running not only the complimentary CMS[3] operating system, but also other available operating systems for the S/360. During development of both the S/360 model 67 and the S/370, a model of these machines were used on a previous version to simulate the machine in development.

2.4.2 Modern background for virtualization

Nowadays many businesses have invested largely in computer equipment and server hardware where their equipment might only use a fraction of their available power and stand idle most of the time. Virtualization emerged as a technology to better use the available resources by enabling several processes and operating systems to coexist on the same server without taking a performance hit.

The emergence of cloud computing is largely helped by the existence of virtualization tools. The usage of one server to comply to several users needs is of course one of the mayor selling points. Important is also the need to consolidate the existing servers, migration of workflows and VMs as well as on-demand creation of VMs[54].

Virtualization also gives a large amount of flexibility to its users, allowing them to use VMs from anywhere. The flexibility also gives application developers and researchers the possibility to develop their own virtual test-beds that can be created when needed[50], and easier to maintain, removing the waiting time for hardware.

The need for virtualization in modern day is apparent with modern technologies such as cloud computing and the increasing capacity of personal computers, which enable regular users to take full use of virtualization at their desktop. The use for virtualization to further lengthen the use of legacy applications and operating systems to lengthen their lifetime and to lighten maintenance cost and complexity. Virtualization technology also has a place in academia and education, making life easier for students to set up their own network topologies and to make it easier for development in operating system courses, as well as other research.

[3]Conversational Monitor System

2.5 A brief history of virtualization

For many, the term virtualization and the concept of virtual machines (VMs) might seem like a new and fascinating concept, although the history of virtualization is almost as old as modern computing itself. The concept of virtualization itself builds upon the concept of paging and virtual memory. The introduction of VMwares virtualization solution, VMware Workstation in the late nineties[50], was one of the first virtualization suites available for X86 based computers. Later products like Xen and the hardware virtualization technology from Intel and AMD has made virtualization to an industry standard of computing.

My short history lesson will focus mainly on the VM system of IBM, before the rise of microcomputers in the eighties. We will then focus on the history of virtualization for the x86 platform and some of the challenges that were involved in virtualizing the X86 architecture.

The sections within the *Early history of virtualization* has been able to write with the help of the following sources.[51, 38, 53, 12, 16, 48, 33, 28] Some research on related Wikipedia articles have unfortunately had to be used, namely these[64, 65]. These have been used due to the nature of some of the articles related to CP/CMS development that is hard to get a copy of, namely[1, 37]. All usage of Wikipedia was cross referenced with the previously mentioned sources and only used to fill in gaps where necessary.

2.5.1 Early history of virtualization

The idea behind VMs originates in the concept of virtual memory and time sharing. All of which are concepts that were introduced in the early 60s, and pioneered at the Massachusetts Institute of Technology and the Cambridge Scientific Center.

Virtual memory and time sharing

Virtual memory and paging is used to allow programs that are larger than the available memory to exist, and most importantly to allow several programs to exist and share the same memory. A crude simplification is to say that virtual memory makes the memory appear larger than it is to programs, and having only the most accessed data present in the systems main memory.

Virtual memory first appeared in the 1960s. One of the first notable systems to include virtual memory was the Atlas Computer, which was jointly developed by the *University of Manchester*, *Ferranti* and *Plessey*. To IBM virtual memory was unproven ground and territory they were unsure to venture into. Partly due to the fact that the virtual memory system of the Atlas computer were not working as well as was hoped for, and no one seemed to know why it was not working. This would turn out to be caused by thrashing, as later explained by research done by the CP/CMS team and developers working with the IBM M44/44X. In 1963 MIT started Project MAC, a project intended to develop the next generation of time sharing systems.

Time sharing emerged in the early 1960s, although papers discussing time sharing had emerged in the late 1950s[45]. The most notable time sharing

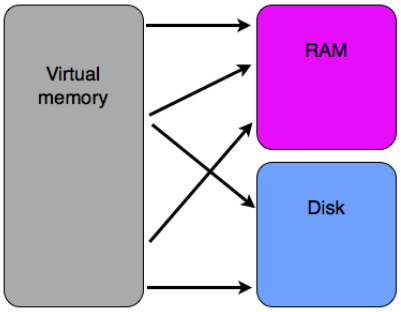

Figure 2.4: Virtual memory abstraction with pointers to RAM memory and the disk.

project took place at Massachusetts Institute of Technology under the leadership of Professor Fernando J. Corbató. This project was to be known as the *Compatible Time Sharing System (CTSS)*. Corbató was also to be one of the most prominent members of Project MAC along with Robert Creasy, which was to become part of the team to develop IBMs first venture into virtual machines.

CTSS first emerged as an idea of John McCarthy who released a memo in 1959 which proposed a time sharing system for an IBM 709 computer at MIT. The design of this computer system started in the spring of 1961 under Corbató. CTSS was first demonstrated in the fall of 1961.[53] And was operational at MIT until the summer of 1973. The idea of time sharing systems was to let users interact directly with the computer. In the early 1960s, the pressure was on IBM to develop a time-sharing system, at the time the most prominent type of computing was batch processing.

The submission to Project MAC that IBM submitted was not as well received as IBM had hoped for, and Project MAC ultimately went with another vendor. The failure of Project MAC for IBM led to the development of CP/CMS, the future virtual machine system for IBM.

IBM System 360

In the 1960s when a computer vendor, such as IBM, released a new computer system, they all started with a new design, this resulted in each system to be designed with a "clean sheet". For many users this approach led to frustration with new technical specifications that needed to be learnt, incompatibilities with existing hardware, such as printers, tape drivers and memory architectures. All software had to be migrated with each new system, which also would be costly and time consuming.

With these issues, IBM took a risky undertaking when they developed and announced the System 360. S/360 was to be backwards compatible, and was to replace, among others, the IBM 7000 series of computers. For the S/360 to be backwards compatible, it meant that there would be less items to change for the users when they needed to change parts or upgrade their system. This was a "mix-and-match" approach, so that each user could tailor their system to suit their specific needs.

To begin with S/360 was not to have any traits of time-sharing systems, such as virtual memory, time-sharing was unimportant to IBM and the problems of the Atlas computer did not help. In February 1964 IBM launched the Cambridge Scientific Center (CSC), this project was launched so that IBM would have an advantage during Project MAC. Their confidence also strengthened since CSC was located in the same building as Project MAC at MIT. IBM had learnt that MIT was leaning towards another option than theirs, that included virtual memory. Since the S/360 had no virtual memory, IBM modified their S/360 but the final product was seen as to different from the rest of the 360 computer line.

CP/CMS and CP-40

With the loss of Project MAC the CSC team was devastated, and to earn back the confidence lost in Project MAC the CSC team decided to build a time-sharing system for the S/360. To lead the CSC team Robert Creasy went from Project MAC to CSC, and began the work on what was to become the CP-40 virtual machine operating system. In tandem staff from the team worked closely with MIT to provide a version of the S/360 that would suit their needs, this was to become the S/360-67.

With work on the CP-40 going on steadily, providing input and research results for the S/360-67 team. The CP-40 was to become a second generation time-sharing system for the newly announced S/360 in 1964. It was to be designed for a wide range of activities operating system research, application development and report preparation. Since the S/360-67 would not arrive for some some time, the CP-40 team modified a S/360-40 to support virtual memory to support development of CP-40 and CMS in the mid of 1965.

The design of CP-40 was inspired by a recommendation from members of the IBM Research team to use the principles of virtual machines to time-sharing planners. CP-40 design was then based not only on the idea of virtual memory but also that of virtual machines[4]. To achieve virtualization of the S/360 instruction set they formed a strategy of using a *supervisor state* and a normal *problem state*. Each virtual machine was to be executed entirely in problem state, and privileged instructions were to be reproduced by the Control Program (CP - the supervisor[5]) in the VM, and certain instruction would be intercepted by the hardware.

The CP-40 used full virtualization to virtualize its guests. Early systems allowed for up to fourteen virtual S/360 environments, all of which were isolated from each other. This allowed its users to simultaneously develop non virtual operating systems on the CP-40 as well as high degree of security. The Cambridge Monitor System (CMS - later renamed to Conversational Monitor System) was separated from the CP to allow for a modular system design, a lesson learnt from the CTSS. CMS was tailored to be used as a guest under the CP, the two together created a single user time-shared system.

[4] Also then called pseudo machines.
[5] What would now be considered a VMM or hypervisor.

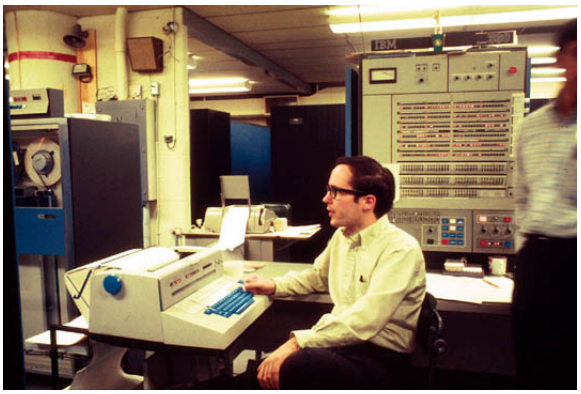

Figure 2.5: An IBM System/360-67 at the University of Michigan. Image courtesy of Wikimedia Commons.

S/360 and CP-67

In 1966 the CSC team began the conversion of CP-40 and CMS to the S/360-67 computer, since the CP-40 was built on a custom built computer system this was a significant re-implementation. The development of CP-67 was not initially done on a real S/360-67 but on a S/360-40 that was modified to simulate a 67. This approach was repeated during the development of the first S/370 prototypes.

Demand for CP/CMS and virtual machine systems came early, this was mainly caused by the frustrations that users of the IBM Time Sharing System 360 (TSS/360) had with the system, which suffered in both performance and reliability. This demand shocked IBM, who had invested a lot in their time-sharing endeavors, and who had already tried to kill off the development of CP/CMS. However, the interest in CP/CMS began to grow, among them Lincoln Labs had expressed interest in the project early on, and shortly after the production of the CP-40 began, CP/CMS was already in daily use at Lincoln Labs.

At the same time interest in TSS/360 began to diminish, and was ultimately killed off in 1971. This helped pave a way for CP/CMS as it came into the light as a viable alternative to TSS/360. CP/CMSs first version was released in may of 1968. CP/CMS was also unusual as it was released as open source to its users, released through the IBM Type-III Library. This meant that on several sites that ran it, ran an unsupported operating system, this also helped to create a community around the S/360-67 and CP/CMS.

VM/370 and the future

In the summer of 1970 the team that had worked on CP/CMS began work on the System 370 version of CP/CMS. CP/CMS turned out to be a vital part of the S/370 development, as it allowed for simulation of other S/370 systems and the S/360-67, this also allowed for development of S/370 before the hardware for the S/370 was available. The first releases of S/370 did not initially support the use of virtual machines, although the addition of the DAT box (Direct

address translation) would allow the use of virtual memory and and finally virtual machines.

Mid 1972 marked the end of CP/CMS and the beginning of VM/370, the code base of VM/370 was largely based on CP/CMS. The VM/370 was now also a real system, and no longer part of the IBM Type-III Library, although it continued to be released through this portal until the mid 1980s.

The VM family of operating systems from IBM has continued to exist, and been used on the System 390 and eventually the z/Architecture. The current VM version known as z/VM, still keeps backwards comparability with the older architectures on which virtualization was pioneered.

2.5.2 X86 virtualization and the future

This section will focus on the history of virtualization on the X86 platform. A processor architecture that was considered nearly impossible to virtualize, however with the X86 processor architecture growing more and more commonplace in the 1990s the quest to virtualize this architecture has proven fruitful.

Challenges with the X86 architecture

From an historical point of view, the X86 processor architecture was never intended to support virtualization. When the architecture was designed it was assumed to only have one user, and was initially not designed to support virtual memory. Meanwhile regular virtualization techniques such as *trap-and-emulate* was commonplace at its time on IBM mainframes running VM/370 and eventually z/VM virtualization systems. As the X86 processor became more and more popular, both on desktops and in server environments, it was becoming evident that virtualization was a highly requested feature to the X86 architecture.

The characteristics of a virtualizable architecture that were defined by Popek and Goldberg in 1974[36], which state that a machine is able to support a VMM if there is at least two modes of operation. A method for non-privileged instructions to call privileged system routines. A memory relocation or protection mechanism, and lastly there should exist asynchronous interrupts to allow the I/O system to communicate with the processor.

The X86 processor met all of these characteristics. It has four modes of operation, which is the four processor rings, as well as methods to control the transfer of programs between levels[6]. There also exists facilities to enable paging and virtual memory. However the X86 architecture was still not able to support virtualization, viz. there exists instructions for the processor that would cause traps and alter the state of processor register. Which in turn could alter the state of the VMM and possibly the host itself, identified by Popek and Goldberg as sensitive instructions. I.e. instructions that alter memory resources or changes processor registers that could affect the processor mode.

Examples of such instructions that would cause these traps was investigated Robin and Irvine in[41], and includes instructions that were placed in two groups; sensitive register instructions and protection system references, the instructions are presented in Table 2.1.

[6] Also known as call gates.

Sensitive register instructions	SGDT, SIDT, SLDT, SMSW, PUSHF, POPF
Protection system references	LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT N, RET, STR, MOVE

Table 2.1: Instructions that cause traps.

Another issue that the X86 platform had to make it virtualizable, was the protection ring mechanism, which does meet the Popek and Goldberg requirements. However, the way these work give rise to other issues. The four rings are present to make the X86 platform a secure environment. Applications and user-space processes run in the fourth processor ring, while the operating system or hypervisor runs in the first processor ring. The second and third rings are not used in modern operating systems.

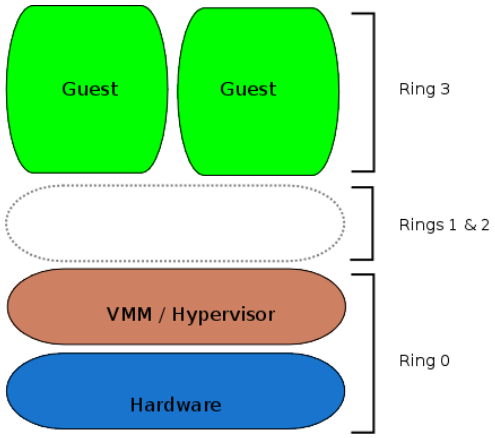

Figure 2.6: Hypervisor and guests with regard to processor rings.

Most modern day operating systems only utilize rings 0 and 3. When running a hypervisor in the first ring, this enables the hypervisor to run at the most privileged level, which also lets it control all hardware and system functions. While guests have to be run in user-space, i.e. ring 0, which means that privilege instructions that are meant to be run in the first ring, actually runs in ring 3. When the guest issues these instructions this will cause a fault inside the guest.

Early attempts at virtualizing the X86 platform would emulate the entire CPU in software, this yields very poor performance for the guests. Others deemed virtualization of the X86 architecture impossible, or in the best case impractical, due to legacy code and architectural quirks.[2] As we shall see early successful attempts at virtualizing the X86 architecture involved either some form of binary translation or modifications to the guest in order to achieve better performance than emulation.

Binary translation and VMWare

Since classical virtualization of the X86 processor architecture in the same way that was done for the IBM System 360 computer was not possible, development of other techniques to enable virtualization was the obvious next step. In 1998 VMWare introduced a VMM that could virtualize the X86 architecture[2]. To make virtualization of the X86 platform a possibility VMWare used a technique called binary translation. This technique lets the guest run directly on hardware, and when privileged instructions that cause traps are encountered they are handled by the hypervisor and emulated. In addition the guests are allowed to run unmodified and unaware of being virtualized.

This works by scanning the guests memory for instructions that would cause traps, in addition to identify instructions that would allow the guest to know it is running in ring 3. When these instructions are found in the guest they are dynamically rewritten in the guests memory. This happens at run-time and the privileged instructions are only translated when they are about to execute, so performance is always at its best. While complex to implement, it allows the guests to yield higher performance as opposed to the performance yielded when being completely emulated. As well as letting guests run unmodified on the hypervisor.

Paravirtualization and Xen

While binary translation proved to be the first steps towards virtualizing the X86 architecture, another approach emerged in 2003 with the Xen project[4]. This project took another approach than what VMWare had done with binary translation. Where binary translation allows the guests to run unmodified, Xen uses modified guests which are aware of their presence on a hypervisor, this technique is known as paravirtualization.

These modifications on the guests were initially developed for the Linux kernel, and subsequently incorporated into the mainline Linux kernel tree starting with the 2.6.23 version. Later on these changes have also been made available as drivers for Windows. At the same time both Intel and AMD released extensions to their respective processors for the X86 architecture to enable 64-bit addressing. Which would address the limitations of 32-bit addressing in the X86 processors, and also greatly increasing the chances for X86 virtualization to become successful.

Hardware assisted virtualization

In 2006 we saw the arrival of Intel and AMDs hardware extensions to allow for hardware assisted virtualization, making binary translation and paravirtualized drivers not required. The technology from Intel being known as VT-x and the technology from AMD initially being known as Secure Virtual Machine (SVM) later renamed to AMD-V[7]. The way this allows for virtualization is to introduce a new operating mode, host and guest. Thus also making it possible to virtualize the X86 platform in the classic trap-and-emulate approach that was well understood in VM use on IBM mainframes such as the S/360 and 370.

[7]The presence of the SVM name is still present as a CPU flag.

Intel VT-x	VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMCLEAR, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, INVVPID, VMFUNC, VMCALL
AMD-V	CLGI, INVLPGA, MOV (CRn), SKINIT, STGI, VMLOAD, VMMCALL, VMRUN, VMSAVE, RSM

Table 2.2: Intel and AMD new and modified instructions for the X86 hardware virtualization extensions.

Both introduced several vendor specific instructions for these technologies which are listed in Table 2.2. In addition both added data structures to store state information about the guests present. Intel naming theirs Virtual-Machine Control Structure (VMCS) and AMD theirs Virtual Machine Control Block (VMCB).

Since the guests cannot directly access memory, the hypervisor needs to provide a virtualized memory for the guests that maps the virtual guest memory to physical host memory. Initially this was implemented in software as shadow page-tables in hypervisors. However both Intel and AMD developed technologies for these memory operations, Extended Page Table (EPT) and Rapid Virtualization Indexing (RVI)[8], to provide a virtualized memory management unit for the guests. Allowing for performance increases as memory can be handled in hardware and not software implementations.

Kernel-based Virtual Machine (KVM)

With the advent of the hardware virtualization extensions the Kernel-based Virtual Machine (KVM) made its appearance from an Israeli technology business known as Qumranet[23]. This technology is a kernel device driver for the Linux kernel, which takes full usage of the hardware extensions to the X86 architecture. Where Xen virtualized guests needed to have drivers or modifications to the operating system, KVM allowed for guests to run unmodified, thus making full virtualization of guests possible on X86 processors.

A goal of KVM was to not reinvent the wheel. The Linux kernel already has among the best hardware support and a plethora of drivers available, in addition to being a fully blown operating system. So the KVM developers decided to take use of the facilities already present in the Linux kernel and let Linux be the hypervisor. Where Xen have had to more or less completely write a new operating system with drivers and a scheduler, KVM simply takes use of the hardware extensions. KVM also allows the guests to be scheduled on the host Linux system as a regular process, in fact a KVM guest is simply run as a process, with a thread for each virtual processor core on the guest.

The future

With Libvirt making its way into the virtualization world, allowing for an open API to tie virtualization services together, and technologies such as oVirt being

[8]Formerly known as Nested Page Tables (NPT)

built upon this as well, a new abstraction to virtualization is possible. Cloud computing has become commonplace, with virtualization technology being the cornerstone. Performance of virtualization technologies and hypervisors are also becoming almost as good as bare metal performance, allowing virtualization to survive and become an increasingly important factor in computing for the foreseeable future.

2.6 Benefits and different solutions

This chapter will look further into the advantages and the disadvantages of virtualization. It will also take a closer look at the different virtualization solutions that exist.

2.6.1 Advantages and the disadvantages of virtualization technology

The following sections and paragraphs will look into the advantages and disadvantages of virtualization technology. This is done from the view of full virtualization, or what is generally perceived as the classic, or standard, type of virtualization.

Advantages

Server consolidation, hardware cost and performance Today many companies have several servers that are dedicated to run only one operating system, or even run only one specific service. This often results in servers having high periods of time were the server is idle, which in turn results in hardware that is only running because of one esoteric service. In many cases these services might be running on hardware that is both costly and hard to maintain.

This is were virtualization might be of benefit in many cases. By replacing several small servers with one larger one, and having this large server run virtualization software to allow several operating systems and services to run side by side on the same hardware. This results in the server having less idle time, since all services share the same hardware resources. The costs associated with the power to run all of these servers will also drop, since the need to supply power to several servers now is gone.

Legacy systems and esoteric services can run on its own virtual machine, since the hardware required by these services now only is an abstraction. The costs associated with maintaining costly and aging hardware will also be a thing of the past.

Isolation One of the first and most important backgrounds for developing virtual machines was the ability to have completely isolated machines, where one users errors will not affect the other users. The following is an example of such an error;

During an operating systems course I took during my bachelor we had recently learned about system commands such as the fork command. When a classmate of mine was trying out the fork command, he wrote a little program and ran it on one of the schools six "login" machines. His program and console suddenly

went unresponsive so he logged on to another machine, and the same happened here. He asked the classmates about what was happening. After a close look at his code, it became apparent that he had "fork bombed" two of the schools servers.

The example goes to show that the error of one user or even a program running on a server, affects the other users directly. This is the motivation for using VMs on servers, where users can have their own VM that they control directly. Should a user make a program which could possibly crash the entire system, he only affects himself.[9]

Education Virtualization brings a major advantage for education, with the use of virtualization software in classrooms and computer labs. Entire labs and expensive testbeds could become obsolete. The usage of virtualization in education brings a great deal of flexibility both for students and for teachers. In a course on networking or operating systems, the use of virtual testbeds instead of having a real testbed with in some cases tens of computers, would make for savings for faculties and easier configuration for system administrators.

With a VM testbed a networking course can construct entire network topologies to teach students network routing and network monitoring inside their virtual environment. For the teacher the process of setting up these testbeds becomes easier, as each student can take use of the same configuration on available lab terminals, or on their own hardware. Classes in information security will also benefit from virtualization technologies, with the minimal risk of the students doing mischief on real computers. The isolation that exists between VMs also gives benefit to security courses as well as database related courses and network courses.[8]

Application development For application developers virtualization tools provides testing tools to help during development and debugging of software projects. Virtualization tools like Qemu and KVM are widely used by Linux developers during their development cycle and testing. These tools gives developers a flexible environment for them to work in. The environment will let itself be created and reconfigured more easily than real hardware. Changing memory sizes for instance is a lot easier with Qemu/KVM, by simply increasing it on the console, same goes for other virtualization tools.

Disadvantages

It might seem like there is a lot of benefits to virtualization technology, as with everything in this world everything has two sides. This section will discuss some of the disadvantages that are associated with virtualization technology.

Physical fault With the cost associated with computer infrastructure, it might not always be affordable to have dedicated servers and filling up several racks with one server running Windows, one running Linux and some running

[9] Although a users erroneous program should only affect him, there are cases where an erroneous program can take down the entire host by affecting the hypervisor. Some fellow students and I managed to crash the dom0 in a Xen host when testing a program in a guest, domU. A short description can be found here; http://sygard.no/2010/03/force-reboot-of-dom0-from-domu-in-xen-server-5-5/

legacy systems and so on. The cost benefits associated with running virtualization technology on one server, and having this server allow several operating systems to run at the same time is beneficial in many cases. However, the risk associated with this should not be taken so lightly.

Having only one server also means that there is also one single point of failure. If the server on which the virtualization software runs goes down or becomes unavailable, it will be a problem for all of the virtual machines.

In these cases one should consider what software that is going to run the virtual machine server, or if it is beneficial to have servers that serve specific needs.

Performance While this is one of the big selling points of virtualization technology in the data-centers, better usage of available hardware and so on. This might also be one of the downsides. When sharing one computer with several users the usage of that computer will get better, and the performance of the service should preferably be the same as running on real hardware.

With several users, the performance can take a hit when the number of users or the number of demanding tasks being run on the system, gets higher. This will off course affect everybody using the system and will likely result in displeasure with the system, with possible slow response and bad performance. To avoid such a scenario one will normally scale the system to its use, so that the performance always meets the need of its users.

Application support Not all applications can be run under a virtual environment. Although the VMM should provide an environment identical to real hardware, this is not always the case. In some virtualization products a generic driver is presented instead of a real driver. Qemu/KVM for instance emulates a Cirrus graphics card, that most operating systems support. If an application needs to use the graphic capabilities of say, Nvidia CUDA, there will be no possibility for this application to run[10]

2.6.2 Virtualization technology and solutions

This section will take a short look at the different virtualization solutions that are available as of today in early 2013. It will first look into the open source solutions that are available, and then a look at some proprietary solutions.

Open source solutions

Kernel-based virtual machine Kernel-based virtual machine (KVM) is a virtualization solution for the Linux kernel on the x86 platform. It takes use of the Intel VT-x and AMD-V technology, to allow for virtualization. KVM is an open source-project that is a kernel module that is supplied with each major community and enterprise level Linux distributions, and has been accepted into the Linux kernel since version 2.6.20. KVM offers an interface, `/dev/kvm`, which a user-space program, such as Qemu uses to communicate with the hypervisor. In most cases, KVM is used in conjunction with Qemu.

[10]With this said, the following paper[19] along with the thesis work of Kristoffer Robin Stokke[44], has had some success on exposing external graphics cards to the guest VM.

Red Hat which previously had focused on Xen as the foundation for their virtualization solution, changed to KVM with version 6 of the operating system Red Hat Enterprise Linux[22]. Red Hat had previously acquired the company Qumranet, the initial developers behind KVM, and after having put their effort behind two hypervisor solution, Red Hat decided to focus on KVM.

Qemu Qemu is not strictly a virtualization tool, as it is characterized both as a process emulator and virtualizer. Qemu in itself is only a emulator, when put together with virtualization tools like KVM, it becomes a virtualization tool and a very powerful one at that. In addition it supports a mix of binary translation and native execution, running directly on hardware. Guests that are run under Qemu need not be modified to be able to run. Interfacing with real hardware, like CD-ROM drives, network cards and USB devices is also supported[6].

For instance Qemu, lets the user easily create network bridges to create small virtual networks, that can be used for development[11]. Qemu is also easily modified to support the virtualization or emulation of obsolete hardware, as described in[40]. Where the developers used Qemu to get access to low-level serial and parallel ports to be able to communicate with the desired hardware.

Xen Xen was developed at University of Cambridge Computer Laboratory, and is now under active development by the Xen community under an open source license, although Citrix acquired the product XenSource in 2007.

The virtualization products of Xen is mostly used on mainframes and server hardware. On most CPUs Xen uses a form of paravirtualization using a special interface to allow modified guest to run. Xen does also support unmodified guests using the hardware-assisted virtualization capabilities presented by Intel and AMD in their processor products.

Emulation of external devices is in fact based on the Qemu project, to allow guests input-output virtualization. Live migration of virtual machines to achieve workload consolidation is also possible. A thorough description of the inner workings of Xen can be found here[4]. Interestingly Xen has become a major part in the commercial virtualization solution of Oracle, although as pointed out by[46], Oracle has made significant modifications on Xen to suit their needs.

VirtualBox VirtualBox is one of the most popular virtualization solutions for desktop computers.[18] Providing a virtualization solution to virtualize the X86 platform, this popular virtualization technology is now developed by Oracle Corporation.

VirtualBox can run multiple guest operating systems under the host. Each of these hosts can pause and resume each guest at will, and is able to take snapshots of each of these guests for backup purposes. Each of the virtual machines can be configured independently and can run in either software emulation mode or hardware assisted mode, taking use of Intels VT-X technology or AMD AMD-V technology.

Hardware emulation is also supported in VirtualBox. Hard disks can be stored as files on the host, which can be mounted as drives in the virtual machine. The same can be applied to CD/DVD drives using ISO images on the host. In

[11]This approach has been used by fellow students in their thesis work on network development in the Linux kernel.

addition VirtualBox emulates ethernet network adapters, which enables each guest to connect to the internet through a NAT interface.

Bochs Bochs is not strictly a virtualization solution, but more of a emulation solution. It is a portable X86 platform emulator mostly used for operating systems development. The reason for using Bochs for operating systems design is; when a operating system being developed crashes, it does not halt Bochs, making it possible to debug the operating system, like inspecting registers, after it has crashed.

Proprietary solutions

VMware VMware was in the late 1990s and early 2000s one of the most prominent suppliers of virtualization solutions. Their product *VMware Workstation* turned heads in the late 90s when they managed to tackle the virtualization of the X86 platform, long thought to be unable to be virtualized. VMware managed this by employing the technique of binary translation, which is described earlier in this chapter.

Present day, VMware supplies products both for server virtualization and desktop virtualization. Most famously VMware ESX and VMware workstation. Since VMwares products pre-dates the time of hardware-assisted virtualization, the hardware extensions does not need to be present for VMware products to be able to run.

Microsoft Microsoft has several products for virtualization, their most known, and prominent tool being Hyper-V. This product was released in 2008 for Windows Server 2008, and is available as a standalone product or as a part of Windows Server. Hyper-V uses what they call partitions to support isolation, within each partition an operating system is allowed to execute. Each partition is handled by the hypervisor, and at least one of the VM/partition instances have to have an instance of Windows Server 2008 running. Each virtualized partition does not have direct access to the real processor, instead it sees a virtual processor, which the hypervisor chooses to expose either the whole processor or only a subset of the processor.

Parallels Parallels is best known for its virtualization solutions for the Mac OS X platform, namely Parallels Desktop for Mac which was released in 2006. This release came at the same time as Apple went from the using the Power architecture for their personal computers to the Intel architecture in 2006.

Parallels has virtualization solutions both for desktop computers and servers. For desktop computers, there is *Parallels Desktop for Mac* as already mentioned and *Parallels Workstation* for the x86 Windows and Linux platform.

For servers there is, *Parallels Server for Mac* which is the only virtualization server software for the Mac OS X platform. *Parallels Virtuozzo Containers* is an operating system level virtualization product, which is designed for large scale homogeneous server environments and data centers. To mention some.

Of the most notable features of the *Parallels Desktop* suite and the *Parallels Workstation* suite is that both of these products contain full GPU virtualization. Which among other things makes the virtual machines able to take use of the host GPU device(s) for games or GPU computing.

Solaris Zones Solaris Zones (SZ), also known as Solaris Containers, is an operating system-level virtualization solution. It presents its users with a secure runtime environment, which allows programs to run isolated in their own zones. SZ is present in newer OpenSolaris based operating systems, such as OpenIndiana and the Oracle Solaris suite of operating systems, among them Solaris Express.

Each system with SZ always has one global zone, under which several non-global zones can be hosted. The underlying operating system instance is the global zone[34]. Each zone has its own name, virtual network interface and storage facilities attached to it, there are no minimal dedicated hardware requirements, the zone also maintains its own dedicated user list. SZ allows for the existence of several virtualized application to coexist on the same operating system instance, while still running in isolation from one another.

2.7 Conclusion

In this chapter we have covered what virtualization is, and the requirements for a machine to be virtualized that were formalized by Popek and Goldberg. We have looked at the history of virtualization. From the inspiration to develop a new time sharing system, that would allow users to interact real-time with the computer, to a virtualization system known as CP/CMS that would allow each user their own virtual machine capable of running an operating system of their choice. We also covered virtualization in a modern sense with X86 virtualization the and re-emergence of virtualization in the late 90s. From there on the variations of virtualization techniques, were covered. Before we then looked at some mentionable advantages and disadvantages of virtualization, and the different solutions that are available, both open-source and proprietary.

Currently there is a lot of exiting research and work being done on virtualization. And with its place inside data centers around the world, and the importance virtualization plays for cloud computing. Virtualization will also be a topic of interest with the emergence of green computing, and also earn its place in the home, as more and more technologies finds their way into consumer electronics.

We will now move on and look closer at some of the modern open-source virtualization solutions we saw in this chapter.

Chapter 3

Virtualization software

3.1 Introduction

This chapter will focus on the various available open-source virtualization technologies. We will take a detailed look at each one, and see what differentiates the technologies from each other. The chapter will conclude with a closer look into their differences and compare them all to each other. For those that are interested in installation instructions of KVM and Qemu, they are pointed to the appendix.

3.2 Qemu/KVM

This section will talk about Qemu and KVM. To begin with I will present KVM, its basic architecture and usage. Then I will present Qemu, how KVM ties into the architecture and basic usage. With Qemu and KVM being used quite closely together, the following section on KVM will mainly focus on the kernel module/hypervisor side of the Qemu/KVM suite. The Qemu section will focus on the user-space tools, as well as focusing on where KVM fits into the Qemu architecture and the difference between the regular Qemu and Qemu-kvm.

3.2.1 KVM

About

KVM, or Kernel-based Virtual Machine, is a kernel module for the Linux operating that allows for full virtualization on the X86 architecture. First introduced in 2006, and subsequently accepted into the Linux kernel tree for version 2.6.20. Then developed by technology company known as Qumranet, that was later acquired by RedHat.

A hypervisor is composed of several components usually having to write a scheduler, memory management, I/O-stack for a new hypervisor, as well as drivers for the architecture on which the hypervisor is targeted at. KVM unlike other hypervisors such as Xen, has focused the implementation on the guest handling of the virtualization, letting the Linux kernel operate as the hypervisor. Since Linux has developed into a secure and stable operating system, as well as having some of the most important features for a hypervisor, such as a scheduler

and a plethora of drivers, it is more efficient to reuse and build upon this rather than reinvent a hypervisor.

Architecture

KVM runs as a kernel module in the kernel, exposed as a device on `/dev/kvm`. The module itself handles all communication between the guests and the hardware. All guests has to be initialized from a user-space tool, this usually is a version of Qemu with KVM support. KVM handles the lower level part of VMs, such as controlling the guest to host switches in hardware, processor registers, MMU and related registers, as well as some registers associated with PCI emulated hardware. How the guest to host switches are handled is specific to the Intel and AMD hardware extensions implemented in KVM.

Each guest processor is run in its own thread that is spawned from the userspace tool, which then gets scheduled by the hypervisor. In reality each guest process and processor thread gets scheduled as any other user process alongside other processes, such as a web-browser, by the Linux kernel. In addition each of these threads can be pinned to a specific processor core on a multi-core processors, to allow some manual load balancing.

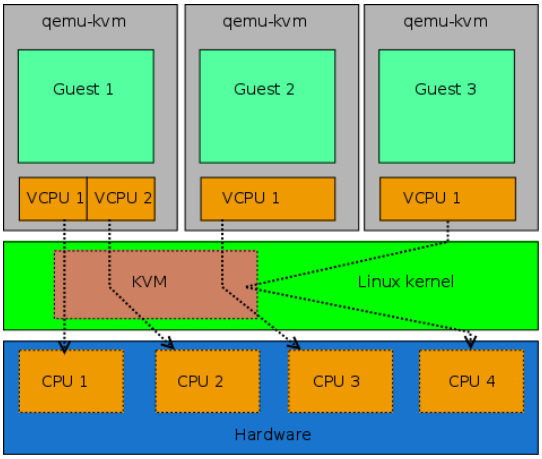

Figure 3.1: The KVM basic architecture.

The memory of a guest is allocated by the user-space tool, which maps the memory from the guests physical memory to the hosts virtual memory, and for the most part handled by the user-space tool. Traditionally this memory translation has been done using what is known as shadow page tables, however they are emulated in software and a cause of expensive exits from guest mode. With the addition of Intels EPT and AMDs RVI technology this can now to some extent be handled in hardware to allow the guest to maintain its own page tables, while the hypervisor only handles its own tables, and the mappings from hypervisor to guest.

Other facilities such as I/O and storage are handled by the user-space tools.

Usage

KVM is never directly used, in most cases of Linux operating systems the kernel module is loaded and present by default. As with all kernel modules KVM can be unloaded and reloaded, upon which a user can add parameters to KVM to enable, among other options, nested virtualization, i.e. `modprobe kvm_(intel or amd) nested=1`, which allows a guest to act as a hypervisor. These parameters are documented in the Linux kernel documentation in the following path in the kernel source code *Documentation/kernel-parameters.txt*.

For more advanced users, such as developers, KVM offers an API using `ioctl`s that allows for communication with KVM. Making it possible to write new software that utilizes the KVM kernel module and hypervisor capabilities of Linux.

3.2.2 Qemu

About

Qemu is a processor emulator[6] that is capable emulating a number of processor architectures, such as X86, both 32- and 64-bit, ARM processors and the IBM System 390. Qemu consists of several components, the CPU emulator known as `Tiny Code Generator (TCG)`, emulated devices such as VGA displays and network cards, as well as user interface and the Qemu monitor.

The Tiny Code Generator is the CPU emulator that takes care of all emulation of a guest processor when hardware assisted virtualization or another hypervisor is used, such as KVM. In essence TCG is a binary translator that performs the translation of a guests instructions for the target CPU. For the X86 processor architecture this was the only way to emulate guests until the advent of hardware assisted virtualization and KVM.

Architecture

The basic architecture for Qemu is similar to what we saw in Figure 3.1. Where KVM handles the lower level part of virtualization, Qemu stands for the emulation that is done and presents the emulated machine to the guest, as well as handling the actual emulated hardware, network interfaces, storage units, graphics, I/O and ports, PCI emulation, as well as some of the memory operations for the guests. It is for that reason not surprising that some ambiguities will exist regarding Qemu and KVM. Especially considering that a guest processor (VCPU) runs as a thread that is launched from Qemu. The memory of a guest is allocated by Qemu at launch, and is mapped into the address space of the Qemu process. This acts as the physical memory of the guest.

Qemu exists as a user application, and runs as a Qemu process being scheduled by the host operating system[21]. This seemingly handles everything that is part of the virtualization and emulation, which is mostly true except when KVM is used for virtualization. Qemu communicates with the KVM module through the `/dev/kvm` interface through a series of `ioctl`s. Figure 3.3 shows the basic flow of the KVM module communication inside Qemu. First opening the interface and issuing the correct `ioctl` to create a new guest. When the guest exits because of a privileged instruction or register modification, Qemu handles the exit and emulates the desired outcome.

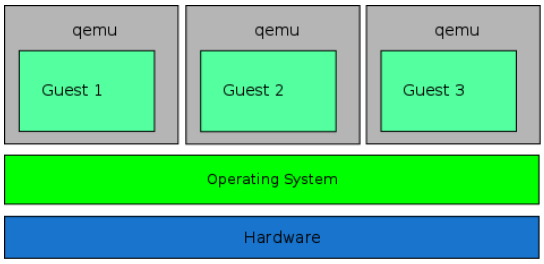

Figure 3.2: Simplified view of Qemu with regard to the operating system.

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
for (;;) {
      ioctl(KVM_RUN)
      switch (exit_reason) {
      case KVM_EXIT_IO:  /* ... */
      case KVM_EXIT_HLT: /* ... */
      }
}
```

Figure 3.3: The basic flow of a KVM guest in Qemu.

Qemu and KVM

For new users of Qemu and KVM the there is some confusion about the the two. To first clarify one thing that should be apparent, KVM is the hypervisor, while Qemu is the user-space tool to interact with the hypervisor. However for many users there is a lot of confusion about the two, this is mainly because the user can choose to use either the regular Qemu command or `qemu-kvm`[1] when they want to use virtualization with Qemu/KVM.

The reason for this confusion is because there were two versions of Qemu available, one from the Qemu developers, that is only known as Qemu. And one from the developers of KVM, known as `qemu-kvm`, which itself is a fork from Qemu[21]. There is not much difference between the two, however the KVM fork was optimized for usage with the KVM module and should be faster. One example of an optimization is the exposure of the host CPU to the guest, using the following command; `--cpu host`, this command is only available for the `qemu-kvm` fork. Another key difference is that `qemu-kvm` only has X86 support, while the other has the possibility to run other architectures.

As of version 1.3 of Qemu, the KVM fork version 1.2.0 has been deprecated. This because from version 1.3, Qemu and `qemu-kvm` has merged their code. Users are advised to use the main Qemu version from 1.3 and on[56].

[1] In some cases this is even just `kvm`.

```
/usr/bin/kvm -M pc-0.12 -m 1024
 -smp 1,sockets=1,cores=1,threads=1
 -mon chardev=monitor,mode=readline
 -drive file=fedora13.img,if=none,
        id=drivevirtio-disk0,boot=on,format=raw
 -device virtio-blk-pci,bus=pci.0,addr=0x4,
        drive=drive-virtio-disk0,id=virtio-disk0
 -device virtio-net-pci,vlan=0,id=net0,
        mac=52:54:00:f5:7a:c9,bus=pci.0,addr=0x5
 -net tap,fd=46,vlan=0,name=hostnet0
```

Figure 3.4: Qemu-kvm command-line example.

Virtio

Virtio was introduced into the Linux kernel to create a series of drivers for Linux that could be used by a virtualized guest and host. Since all virtualization platforms for Linux at the time had their own block and network drivers to mention some, all had various degrees of features and optimizations. With KVM also gaining popularity and not supporting a paravirtual device model, Virtio emerged as a solution to all of these issues[43]. Implementation specific details about Virtio can be reviewed in the cited article.

As Virtio acts as a device driver and model for guests and hosts, it is easily used in conjunction with Qemu. Virtio supports block devices, network drivers and PCI devices, all of which can be triggered from the Qemu arguments. An example of this can be seen in Figure 3.4. Additionally Virtio is used by default in Libvirt and VirtualBox, that we will look closer into later.

Usage

Qemu is usually used through the command line in Unix environments. Usually it is invoked by issuing the `qemu-system-x86_x84` command, depending on what architecture you are targeting or Qemu has been built for. That command is then followed by a number of parameters, such as memory size, number of SMP processors, disk configuration, network configuration and so on.

Often it is sufficient to supply memory and disk configuration to the Qemu parameter list. However this will depend greatly on the users needs, as we can see in Figure 3.4, to which I might add is not complete[20], more advanced parameter configurations is possible, giving the user full control of the virtualized hardware.

For users that find typing a long list of commands for each initialization of Qemu cumbersome, the `-readconfig` and `-writeconfig` parameters can be used. The commands reads and writes configuration files which are utilized to store VM information. The information stored in these files does however not store all information, disk and CDROM configurations are stored here. While memory and number of SMP cores are not stored in this file and will have to be typed in at initialization.

While the amount of possibilities when managing guests, might be daunting for some users with Qemu, it is also the strength of Qemu. That allows the user to take full control of the virtualized hardware and configure everything in the

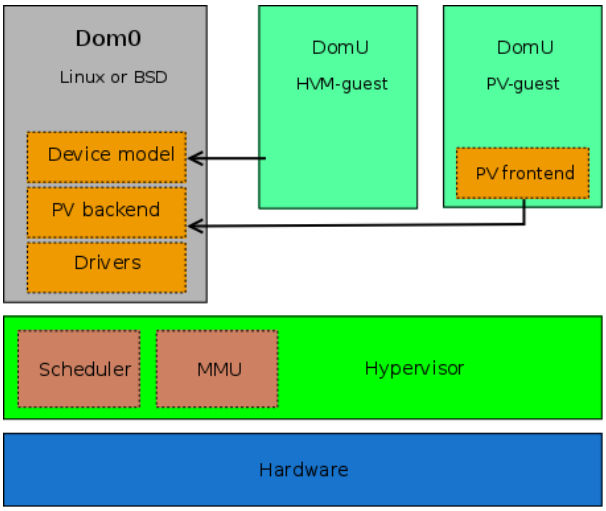

Figure 3.5: Xen architecture with guest domains.

way that the user wants. And even add new emulated hardware to Qemu, to extend the lifetime of applications that might be dependent on specific hardware and so on, as seen in[40].

3.3 Xen

The Xen virtualization suite first saw light in 2003[3, 4], and originated as a research project at the University of Cambridge. The Xen hypervisor is very different from Qemu and KVM architecturally. Where KVM is simply a kernel module for Linux which uses the host Linux system as a hypervisor, Xen is itself a hypervisor. This section will talk about the Xen architecture, how it works and the basic usage of Xen as a hypervisor.

Architecture

Xen uses a hypervisor that runs directly on top of the hardware, the hypervisor is responsible for scheduling and memory partitioning for the guests. All execution of guests are handled by the hypervisor, however the hypervisor handles no I/O, has no knowledge of physical drives, and does not handle any peripheral device. All I/O resources, disk storage and other typical operating system resources is handled by a special virtual machine known as domain 0, or dom0 for short. This is typically a modified Linux kernel or another UNIX system such as NetBSD which can also be used, and is required for Xen to be able to execute any guests. Unlike the architecture of KVM which only adds the hardware virtualization support to the kernel, Xen runs directly on the hardware using Linux or BSD as a mediator that handles, among other things, I/O and storage.

The guests that run on a Xen hypervisor is known as domain U, or domU, and are treated as regular virtual machines. They are scheduled alongside dom0, making for a fairer usage of the processor among the VMs since the host is treated equally with the guests. The present Credit scheduler in the Xen

hypervisor[58] by default assigns the dom0 and domUs an equal weight. Legal scheduler weights is in the range of 0 to 65536, where a domain with weight 65536 will get the most CPU time. By default the weight is 256 for dom0 and all domUs, giving them all equal priority.

Historically the Xen hypervisor only utilized paravirtualization to support guests. With the addition of hardware extensions to support virtualization on X86 processors, Xen is now able to support fully virtualized guests that require no special drivers. The paravirtualized guests has to use special drivers that makes them "aware" of being virtualized, which before the days of hardware assisted virtualization meant a significant increase in performance over binary translation and emulation. With full virtualization the guests can run unmodified and unaware of being virtualized. To achieve full virtualization Xen employs a device model that presents the emulated machine to the guest. This is done by using a stripped down version of Qemu that handles I/O, storage and ultimately emulates the hardware that guests use.

Usage

Xen is usually used through a Linux or Unix based operating system that acts as dom0, through here the interaction is mostly done through commands to the hypervisor using the `xl` command, previous versions used the `xm` command in conjunction with `xend` (Xen daemon). The `xl` tool takes commands as parameters that tell the hypervisor what to do. To start a VM the user will usually use the command `xl start guest.cfg` where the configuration file contains all information about the guest, similar to the command line arguments given to Qemu.

All guests to be launched are all configured through a file known as `xl.cfg`, that contains all information about guests, domain name, number of processors for the guest (VCPUs), memory, disk configuration and network. Examples of these configuration files can be found in the appendix that have been used for the Xen guests that were benchmarked in a later chapter. Subsequently, guests are managed through commands given to hypervisor.

3.4 Libvirt

Libvirt is an abstraction layer and toolkit for managing and administrating virtual machines. It consists of an API and a daemon known as `libvirtd` or simply the Libvirt daemon, which runs on the host computer and listens for requests that are to be mapped to the appropriate hypervisors.

The way Libvirt works is to have a server-client architecture where a server runs the Libvirt daemon, and client applications utilize the API to communicate with the server. Figure 3.6 presents a simplified view of the overall architecture, with user applications that utilize the Libvirt API on the top, communicating with the daemon, which then maps requests to the appropriate hypervisor.

Applications such as `virsh` and Virtual Machine Manager, as we shall see later, utilize the Libvirt API to allow users to communicate with the hypervisor. The API has bindings for most popular programming languages such as Python, Perl, Ruby and Java, mostly used is Python, in which Virtual Machine Manager is written. This allows the API to be used by system administrators to script and

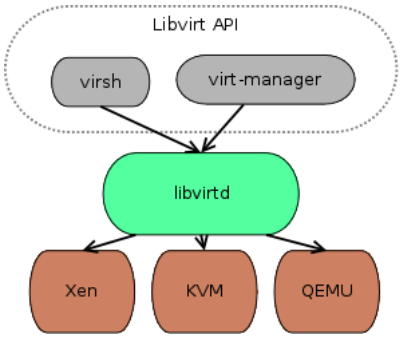

Figure 3.6: Libvirt with regard to hypervisors and user tools.

simplify management of virtual machine. I.e. this can be used to automatically migrate VMs when the total CPU load of a server reaches a certain level so as to keep the load below a critical level.

The daemon which the clients communicate with is the server side component of Libvirt. The daemon listens for requests on a local socket. Incoming requests from clients are forwarded to the appropriate driver for the requested hypervisor and then handled. Libvirt daemon is responsible for starting and stopping VMs, handling the network configuration for the VMs, and migrating guests from one host to another when this is requested.

For Qemu and KVM hypervisor instances Libvirt utilizes either the `qemu-system-x86_64` binary or `qemu-kvm` respectively to handle the virtualization and emulation. Both are looked for in the `/usr/bin` directory, with respect to either using the Qemu emulator or KVM hypervisor when selecting this during configuration. For Xen deployments, Libvirt looks for a running instance of the Xen daemon (xend) and checks for VMs in the `/etc/xen` directory. In addition Libvirt by default uses the Virtio drivers for the KVM guests that are installed.

Libvirt support a number of hypervisors, KVM and Xen are the best known since they are supported by the graphical front-end Virtual Machine Manager. However Libvirt supports, in addition to the aforementioned hypervisors, Linux Containers, OpenVZ, User Mode Linux, VMWare ESX and GSX, Microsoft Hyper-V and IBM PowerVM. The Libvirt API is further used by oVirt to handle cloud deployments of VMs.

3.4.1 User tools and usage

This section will shortly cover the user-space tools for Libvirt, which will be presented in the following order, the `virsh` command line utility, Virtual Machine Manager, Virt Install, Virt Clone, Virt Image and Virtual Machine Viewer. Of which `virsh` is a standalone application supplied with a Libvirt installation. While the last four are supporting tools to the Virtual Machine Manager application.

virsh

Libvirt offers a command line tool to administer virtual machines and the hypervisor, `virsh`. It can be used in two ways either like a typical UNIX command or

interactively. Using this like a command follows the usual syntax for command line applications with parameters, example below.

```
virsh start fedora
```

Which incidentally starts and boots a guest with the name *fedora*. The same commands that can be given as parameters can also be used as commands when `virsh` is used interactively.

The plus side of the `virsh` command lies in its functionality. It supports full control of the hypervisor, fine-grained control of VMs and their hardware features, and live migration of the VMs. However the downside is in usability, where administrators well ventured with the command line will feel at home, those that are more comfortable with GUIs might have a steep learning curve. However there is a tool for them as well.

Virtual Machine Manager

Virtual Machine Manager, or virt-manager, is a graphical front-end to allow users to interact with VMs. The GUI presents the user with information about running VMs, with graphs of CPU usage, used memory and the general status of VMs. The application is written in Python and takes us of the Python bindings that are present in Libvirt to communicate with the underlying hypervisors. At the time of writing virt-manager only supports guests on the KVM or Xen hypervisor.

`virt-manager` presents a clean and understandable graphical user interface to the user that makes it easier to administer VMs on the hypervisor of their choice. Figure 3.7 presents the steps to setup a new guest using a disk image or CDROM. The sixth image presenting the main screen of virt-manager with status of present and running VMs.

Where the `virsh` command line utility can be daunting for some users, the user interface of virt-manager is clean and easy to understand. As seen in figure 3.7, creation of VMs is done in (almost) five steps[2], and administration just as easy.

Guests that are installed using virt-manager are set up, at least with KVM, options that should improve performance, such as Virtio for disks.

Virt Install

Virt Install, or virt-install, is a command line utility for creating new VMs using either the KVM or Xen hypervisors in addition to Linux container guests (LXC). Guests can be installed by text, or graphically using VNC. Specifics to the VM to be installed are given as parameters to virt-install, such as number of virtual cpus, memory, and where to install. Figure 3.8 presents the command to a Linux guest under KVM with Virtio disk support, graphic install using VNC and installation from host CDROM, example is from the virt-install man page.

Virt Clone

Virt Clone, or virt-clone, is a command line utility to clone an existing VM that is connected to Libvirt. The VM to be cloned will be copied in its entirety,

[2]The user might have to configure disk images of LVM partitions for the VM.

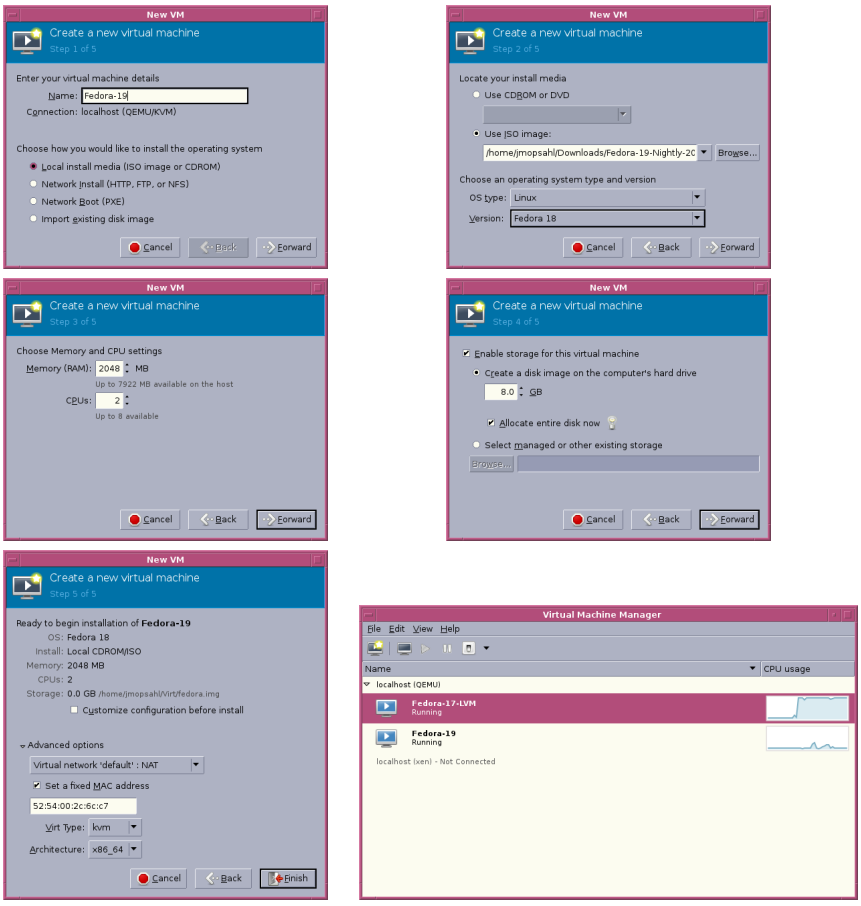

Figure 3.7: Guest creation in virt-manager.

```
virt-install \
        --connect qemu:///system \
        --virt-type kvm \
        --name demo \
        --ram 500 \
        --disk path=/var/lib/libvirt/images/example.img,size=8 \
        --graphics vnc \
        --cdrom /dev/cdrom \
        --os-variant fedora18
```

Figure 3.8: Guest installation using virt-install.

with all hardware configurations being identical between the clones. Uniqueness issues are handled in the cloning process, such as MAC address and UUID.

Virt Image

Virt Image, or virt-image, is a command line tool for installing VMs from a predefined master image. This image describes all requirements for the VM to be installed. virt-image parses the XML descriptor file that is given as a parameter and invokes virt-install to handle the installation. The descriptor file contains requirements that the guest has to the host platform, of which the boot descriptors are most important options, at least one boot descriptor has to be present in the XML file for the guest to be bootable.

Virtual Machine Viewer

Virtual Machine Viewer, or virt-viewer, is a lightweight interface for communicating graphically with VMs. The VM is accessed through VNC or SPICE, and can connect to guests either through Libvirt or via SSH. Examples below using SSH tunnel with both Qemu and Xen:

```
virt-viewer --connect qemu+ssh://user@host.example/system 'VM name'
virt-viewer --connect xen+ssh://user@host.example/system 'VM name'
```

Figure 3.9: virt-viewer commands.

3.5 VirtualBox

VirtualBox is a virtualization application offered by Oracle, formerly offered by Sun Microsystems[3], that can be installed as an application by the user. VirtualBox differs from some of the previously mentioned solutions as it is primarily targeted at desktop users and workstations.

3.5.1 About

VirtualBox only allows full virtualization of the guests it runs, however they can all be virtualized by either software virtualization or hardware virtualization. Guests that are run using hardware virtualization are using the hardware assisted virtualization extensions to the processor to achieve full virtualization. VirtualBox takes full usage of both Intel VT-x and AMD-V[31].

While the software virtualization features are not recommended, this type of binary translation was for early versions the only available virtualization technique for VirtualBox[14]. It uses a kernel driver that runs in processor ring 0 and can handle most instructions natively. The privileged instructions that cause guest exits can then either be recompiled or will be run in ring 1, from which the hypervisor will take over control of the instruction.

Architecturally VirtualBox uses a kernel module that acts as the hypervisor, on top of which the VirtualBox API communicates with the hypervisor. From the API the end-user applications that is the default GUI and the `VBoxManage` command line tool can be built. The architecture that VirtualBox then builds upon is then not so different from what we have seen in other mentioned virtualization suites.

[3]Which in turn had acquired VirtualBox from Innotek the original authors of VirtualBox.

The source code of VirtualBox has borrowed some of its code from Qemu[52], by using some of the virtual hardware devices that can be found in Qemu. In addition they have incorporated the recompiler, which VirtualBox only utilizes as a fallback mechanism when the VMM is unable to handle the situation. Lastly VirtualBox has also utilized the Virtio driver and incorporated it for use as a network adapter when run on a host with Virtio support[32].

3.5.2 Usage

What differentiates VirtualBox from Qemu/KVM and Xen, is that it is available for Windows operating systems, Mac OS X and of course Linux. For Windows and Mac there is a graphical installer that guides the user through the installation, while VirtualBox packages are available in most popular Linux distributions, if not it can be installed from precompiled packages or built from source code.

As is the case with Libvirt and its supporting tools VirtualBox can be used either from command line or from the graphical front-end that is supplied with all installations. The graphical interface is your basic point and click interface, with all possible configurations available, as well as image creation and snapshots.

The command line tools are used as typical UNIX commands. The most important commands are: `VBoxManage` and `VBoxHeadless`, while other commands exists these are the ones that will be used mostly. `VBoxManage` administers VMs in a similar way to Libvirts `virsh` command, however not interactively. While `VBoxHeadless` is a command that lets the user manage VMs without launching the graphical user interface on the host, this can also be used with the `VBoxManage` command with the parameter `--type headless`.

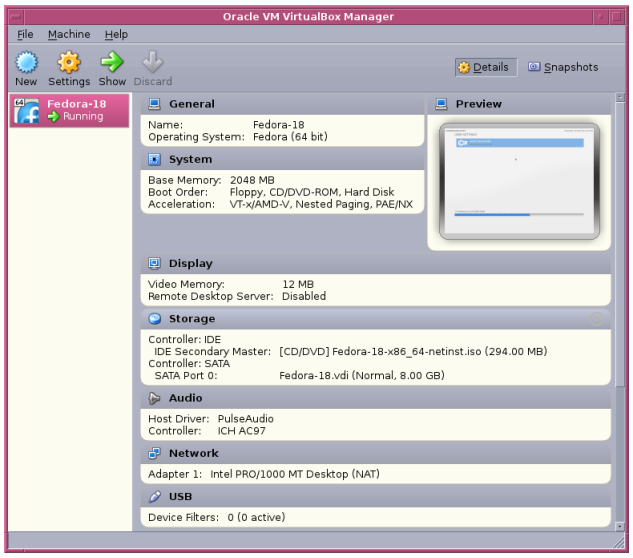

Figure 3.10: VirtualBox main screen.

3.6 Comparison

This section will comment upon the various virtualization suites that we have seen in the previous sections, and will talk about how they compare to each other in terms of architecture and most importantly usage.

Architectural comparison

From an architectural standpoint the virtualization suites that we have looked into have their differences and similarities. Qemu and KVM tie into each other to achieve full virtualization, and form a very powerful virtualization suite that is easy to use and extensible. KVM uses a loadable kernel module in the Linux kernel which reuses the key parts of the operating system to turn the operating system itself into the hypervisor. This lets the user use the host machine while having VMs running on the same system, having the VMs coexist with desktop applications.

While Xen on the other hand has a more intrusive architecture, replacing parts of the operating system and placing the Xen hypervisor itself on top of the hardware. The host operating system is then run as a virtual machine that is scheduled by the hypervisor. Which is quite useful when managing VMs in a server environment, letting the hypervisor have full control of the hardware. Only using the host domain (*dom0*) for administration of guests, and also allowing for a fairer use of the processor. For desktop users that want to use virtualization tools alongside their applications, this architecture might slow performance of desktop applications down for the users.

Libvirt on the other hand builds upon both Qemu/KVM and Xen, putting an API on top of the virtualization suites that eases management of VMs substantially. The API has a command line application built in to administer VMs, the strength comes in the possibility to build applications on top of this API as well, such as Virtual Machine Manager.

Lastly VirtualBox has an architecture that is similar to Libvirt. With a kernel module like KVM that is the hypervisor, on top of which there is an API that allows for management of guests. This also makes for a non-intrusive architecture that benefits the desktop users.

Usage

In terms of usage the virtualization suites that have been covered can be placed in two groups, using command line or graphical user interface. All of the mentioned suites have the possibility to be managed through the command line. For Qemu/KVM and Xen this is the only way to administer VMs and the hypervisor. Qemu has near endless possibilities when it comes to parameters, running a guest can be as simple as supplying only the storage medium. For my benchmarking that will be covered in the following chapters the Qemu command has been this: `qemu-system-x86_64 -m 2048 -hda disk.img -k no -smp 1 --enable-kvm -nographic --redir tcp:2222::22`. Qemu also consists of a monitor that can be accessed when guests are running, from here the guests can be suspended, the CDROM can be ejected, debugging of the operating system is possible, in addition to live migration of the guest.

Xen handles the administration of guests in a quite different way than Qemu, all configuration of VMs is done through configuration files known as `xl.cfg` files, an example is given in Figure 3.11. These files are given as parameters and read by the `xl` tool-stack, from which the basic administration is handled. Basic commands to the `xl` tool is for the most part start and shutdown of VMs, however live migration can also be handled through the same tool.

```
builder = 'hvm'
memory = 2048
vcpus = 2
name = "Fedora-HVM"
vif = ['bridge=virbr0']
disk = ['tap:aio:/var/lib/xen/images/xl-fedora-xen-hvm.img,xvda,w']
boot = 'cd'
keymap = 'no'
on_reboot = "restart"
on_crash = "restart"
```

Figure 3.11: xl.cfg file for a Xen-HVM guest.

While Libvirt does not handle any virtualization directly it does support user space tools that tie the two aforementioned virtualization suites together, and makes it possible to administer them both through the same software. The `virsh` command line tool and the associated command line tools, make it possible to easily install and administer VMs without having to learn about the quirks and all available options of either Xen or Qemu. In addition the Libvirt API allows for other applications to be built that can use the underlying virtualization software, one of these tools is Virtual Machine Manager that lets users graphically manage VMs on their hypervisor of choice.

Lastly VirtualBox that by default comes with a graphical user interface is probably the easiest to use, and requires the least of the user. Being available on both Windows and Mac OS X in addition to Linux also makes this a familiar tool for users across operating systems.

	Qemu/KVM	**Xen**	**Libvirt**	**VirtualBox**
Host OS	Linux	Linux, NetBSD, OpenSolaris[59]	Linux	Linux, Mac OS X, Windows
Guest OS	Linux and variant, BSD and variants, Solaris, OpenSolaris, Windows 7, Win. 8, Win. Server 2008, Win. Server 2003, Win. Vista, Win. XP, Win. NT[55]	Linux and variants, BSD and variants, Windows 7, Win. Server 2008, Win. Server 2003, Win. Vista[60]	Linux and variant, BSD and variants, Solaris, OpenSolaris, Windows 7, Win. 8, Win. Server 2008, Win. Server 2003, Win. Vista, Win. XP, Win. NT[55]	Linux and variants, BSD and variants, Solaris, OpenSolaris, Windows 7, Win. 8, Win. Server 2008, Win. Server 2003, Win. Vista, Win. XP, Win. NT, Mac OS X[57]
Full virtualization status	Yes	Yes	Yes	Yes
Paravirtualization status	Yes, with Virtio	Yes	Yes, with Virtio for Qemu/KVM and Xen-PV	No
Scheduler	Host OS	Own	Depends on VMM	Host OS
Command line tools	Yes, as parameters to Qemu	Yes	Yes	Yes
GUI	Yes, with Qemu and SDL, and Virtual Machine Manager	No, available through Libvirt with Virtual Machine Manager	Yes through Virtual Machine Manager	Yes
Snapshots	Yes, using `qemu-img` tool	Yes, through `qemu-img` or LVM snapshot facilities if using LVM	Yes, using `virsh`	Yes
Live migration	Yes	Yes	Yes	Yes
Editable configuration file	Yes, with `-readconfig` option	Yes, default	Yes	No, GUI or `VBoxManage`
PCI-passthrough	Yes	Yes	Yes, KVM	Yes

Figure 3.12: Comparison of the various virtualization suites.

Chapter 4

Benchmarks

4.1 Introduction

This chapter will present the benchmarks that will be performed in this thesis. To begin with I will present the articles that have been the basis for the benchmarks, and that have influenced the choices done. I will the present the various virtualization platforms to be tested in addition to some technicalities about these, before we look at the various benchmarking suites that are to be used for the benchmarking. This chapter ends with the experiment design of the benchmarks.

4.2 Motivation and previous work

This section will present the articles and papers that have inspired the benchmarks. Each paper will be presented shortly, and this section will conclude with a summary that explains the various choices done and why the articles have been interesting.

Performance Measuring and Comparing of Virtual Machine Monitors

Che et al.[10] performed benchmarks of both Xen and KVM, using Linpack, LMBench and IOZone to perform the benchmarks. Using hardware with an Intel Core2DUO processor running Fedora Core 8 and kernel 2.6.24. The versions of Xen and KVM that were tested have both been surpassed since then, using Xen version 3.1 and then KVM-60, KVM is now updated with each kernel release.

Their findings with Linpack, not surprisingly favored bare metal performance before Xen, while KVM was substantially lower. The lower performance of KVM is explained by KVM having to check every executing instruction, and deciding whether or not to stay in guest mode. The Xen performance is explained by Xen executing every floating point operation on the CPU. Their memory based benchmark suggested that all virtualization suites performed similarly on memory read operations, while for write operations KVM performed worse. Context switching was also measured and suggested that KVM performed the worst, while the host performed the best.

The last benchmarks presented tested the file system by using IOZone, this tested both read and write operations with a record length of 1024 kilobytes with sizes ranging from 1 to 64 megabytes. The results favored Xen using Intel VT-d technology that surpassed KVM performance by factor of six.

Since the paper was published in 2008 it would be interesting to see the status of the virtualization suites, given the rapid development of Linux and both Qemu and KVM. The then version of KVM did not support Intel EPT or MMU optimizations, giving rise to further overhead of switching between guest and host mode.

Quantifying Performance Properties of Virtual Machine

Xu et al.[67] benchmarked Xen, KVM and VMWare, and tested the overall performance, isolated performance and scalability of the VMs. The benchmark was performed on a Intel Core 2 processor with CentOS 5.1 with kernel 2.6.24, on which Xen 3.2, KVM-62 and VMWare Workstation 5.5 was tested.

The first benchmarks tested the performance of the CPU and memory using UBench[35], of which Xen performed the best. Fork operations tested favored VMWare, using a customized program. While gzip compression favored KVM, and LAME encoding and decoding favored Xen.

For disk and file systems testing on overall performance IOZone was used with file size 64 megabytes and increasing record length. The results using read operations indicated that VMWare performed the best. Write operations had similar results for all virtualization suites. Of both read and write the host performed the best.

Like the previous paper this was also published in 2008 using similar hardware and slightly newer versions of virtualization suites. The overall performance results favored Xen in most cases. Newer benchmarks of raw CPU performance, memory operations and disk tests, can give an indication of whether or not Xen is still the best performer.

A Study of a KVM-based Cluster for Grid Computing

Presenting a performance study of how KVM would perform in a Virtual Organization Cluster for grid computing, Fenn et al.[15] used a 16 node physical cluster on which they ran CentOS that in turn was chosen as the OS for their virtual compute nodes. The main focus of the paper was to see how KVM would fare as a hypervisor in a virtual cluster. In addition to KVM the authors tested performance on Xen and of course physical nodes. KVM version used was KVM-77, and the benchmarking suite used was the High Performance Compute Challenge which include the High Performance Linpack benchmark.

Although the authors had issues with network latency when using KVM, they concluded that KVM was generally efficient when the network is not a factor. The reported results show that KVM was slightly surpassed by Xen when run on a single node. When run on all nodes KVM was largely surpassed by the host and Xen, however having 2-3 times higher latency than the physical nodes and Xen.

A Synthetical Performance Evaluation of OpenVZ, Xen and KVM

In this paper[11] Che et al. have tested three virtualization suite that all utilize different virtualization techniques. Namely OpenVZ which utilizes operating system level virtualization, Xen with paravirtualization and KVM with full virtualization. Performance measuring was done on an Intel Core2DUO processor running Gentoo Linux 2008 with kernel 2.6.18. OpenVZ was patched into the kernel version, Xen ran version 3.3 and KVM version 75. Interestingly the paper, which was published in 2010, uses versions of software that were common in 2008, the Gentoo version was old at the time. The version of KVM was released in 2008 as well and not originally developed for the 2.6.18 kernel, as it was developed for kernel version 2.6.27 and higher.

The benchmarks measured both macro and micro performance of the virtualized environments with several benchmarking tools. Among them we find Linpack, LMBench and IOzone as seen in[10]. The Linpack results favored the host over the virtualization suites, with OpenVZ and Xen closely by. The LMBench results show equal performance for read operations, while the write operations show that the virtualization suites outperform the host when the size is larger than the available cache sizes. The IOZone results also reporting Xen as the best performing virtualization suite, interestingly they report a decrease in performance for read operations when run with multiple cores, and the reverse for write operations. One of the reported micro benchmarks was LMBench context switch, the reported results presented OpenVZ as the fastest of the suites, with Xen and KVM having a substantial degradation.

From the results presented in this paper it would be interesting to see if newer versions of operating system and virtualization suites have matured, and what performance would be reported today.

Recommendations for Virtualization Technologies in High Performance Computing

In the following paper, Regola and Ducom[39] evaluated the three open-source virtualization suites, OpenVZ, KVM and Xen, with regard to workloads when running HPC jobs, with OpenMP and MPI. Although this article is slightly outside the scope of this thesis it presented some interesting findings that directly attributed to the some of the benchmarking design choices.

For the disk benchmarks, the authors focused on OpenVZ and KVM versus native performance using IOZone as a benchmark. From which the results favored KVM over OpenVZ, and even the host, in read performance. While in write performance KVM was largely surpassed by OpenVZ and the native performance. The authors also tested KVM using various disk configurations for the guest, among them using a host partition directly, block level. As well they tested the RAW, Qcow2 and VMDK disk image format, of which VMDK performed the best for read operations, while Qcow2 performed the best for write operations. The results were different with writeback and writethrough.

The paper being published in 2010, it would to see if performance on various virtualization platforms is different with regard to their disk configuration.

The Evaluation of Transfer Time, CPU Consumption and Memory Utilization in Xen-PV, Xen-HVM, OpenVz, KVM-FV and KVM-PV Hypervisors using FTP and HTTP Approaches

In this paper Tafa et al.[47] evaluates the performance of both para- and full virtualization of both Xen and KVM, in addition to OpenVZ. This is done using a customized benchmark that the authors have created themselves. To benchmark the virtualization platforms they test CPU consumption and memory utilization, through HTTP and FTP approaches.

The results favor the Xen-PV suite over the other benchmarked platforms in CPU performance, with Xen and KVM utilizing full virtualization in the other end. For their memory utilization results the authors concluded that Xen-PV performed best, while Xen and KVM with full virtualization performed the worst.

This paper has been interesting due to the hardware that the benchmark has been run upon, which is a Intel i7 920 processor. That is an older model of the processor in the computer available at laboratory which my tests are to run on. In addition it has introduced the idea to test both full and paravirtualization of Xen and KVM in my benchmarks.

Quantifying the Cost of Context Switch

In this paper Li et al.[26] presents a method to measure the cost of a context switch on a Linux system, with 1 or more cores present. The authors use different sizes for processes to be switched, a varying number of processes and an optional strided access pattern of the process data. Their implementation to measure the time of a context switch will be utilized in my benchmark, a more thorough description of the measurement approach will follow.

The reason for basing a benchmark on the presented approach is because other methods to measure context switching, such as LMBench, are not 100 percent accurate. For that reason I want to get results from two different approaches to context switches.

4.2.1 Summary

The design of the benchmarks in this thesis and the configuration of the test setup, has been largely influenced by the articles mentioned above. The choice to use the context switching application from Li et al.[26] is attributed to the the context switch measurement done in these papers [10, 11]. Of which it would be interesting to see how the context switch time of virtualized applications has changed since these papers were published, especially with the advent of hardware assisted page tables. As LMBench measures context switches with a 10-15 percent error margin, it would also be interesting to see if these two approaches to measure context switches will yield the same results.

In addition to context switches, I will perform the Linpack benchmarks from [10, 11] and [15] in my benchmarks. All of these articles show results that favor either OpenVZ or Xen over KVM, here I would investigate if the performance presented in those papers still holds true with all development that has been done, especially with regard to KVM.

Memory benchmarks are largely inspired by Che et al. and Xu et al. in [10, 11], of which I will perform the same memory benchmarks. And see if their

findings are still relevant with the development that has happened over 5 and 3 years respectively.

To test the file system and disk configuration I will use IOZone as have been done in [10, 67, 11] and in [39]. All of the mentioned articles present results in which Xen performs the best. With a basis in these articles I will measure the performance of Xen, KVM and VirtualBox, to see what the current performance of the virtualization suites is. In particular with the advent of Virtio since the measurements done in these papers. In addition Regola and Ducom[39] measured the disk performance of KVM on multiple disk configurations, the same will be done for my benchmarks, which will utilize the most common and popular formats.

The measurements done in [47] presented some interesting results, however I will not replicate their measurement approach. I will take inspiration from their configuration by measuring the performance of full- and paravirtualization of the virtualization suites of on those support these virtualization paradigms. I.e. Xen-PV and KVM with Virtio.

The benchmarks that I will perform are largely inspired by the papers that I have presented, mainly with regard to the benchmarking tools that have been chosen. From the results that the papers have presented I would investigate if the development of KVM has lead to it performing better than what Xen does.

4.3 Virtual Machine Monitors

This section will highlight each of the virtual machine monitors (VMMs) to be tested and the reason for testing specific VMMs. In some cases, several VMMs that are similar will be tested, e.g. Qemu/KVM. This is done to rule out any minor differences among VMMs that are developed by the same developers, meanwhile also highlighting their differences that might incur overhead.

Common for all guests are that they run with 2048MB/2GB of memory. For guests where we can expose the host CPU directly we will do this, in addition the default CPU will be tested. Where these settings are applicable, this will be specified in the following. When it comes to SMP support we will test both one, two, four and eight cores on the guests to see how this fares with the guests. Some of the suites will be compiled from source, namely Qemu and KVM. In some cases they will need additions to the configuration, others will run the default configuration. Specific details on compilation and requirements are referred to in the appendix.

VMMs / Hypervisors	
Libvirt/Virt-manager	KVM
	Xen-PV
	Xen-HVM
QEMU/KVM	QEMU-main branch
	QEMU-KVM branch
Xen (`xl`)	Xen-PV
	Xen-HVM
VirtualBox	VirtualBox

Table 4.1: Table showing the various hypervisors to be tested.

4.3.1 KVM

Many of the tested VMMs makes use of KVM in one way or another. As all Linux kernel comes with KVM 'pre-installed', we only know which kernel version which the KVM module is targeted for. To make sure that we are consequent with the KVM version, the version used is `kvm-kmod-3.6`. This is a standalone kernel module that can be installed on all Linux systems, however the user has to compile it first and install it himself[1]. Installation instructions for **kvm-kmod** follows in the appendix.

4.3.2 QEMU

We test the official release of Qemu release, which comes from the main source tree. It has only been developed by the Qemu developers, with additions from the KVM developers branch that has found its way back. It is compiled with only one target, x86_64-softmmu, this to minimize compile time, in addition we have no need to test the other targeted architectures in Qemu. In addition the compiler flag to enable use of KVM is also used, as well as a flag to disable the use of the Tiny Code Generator (TCG), this is done because we only want to test the hardware virtualization of Qemu, not the emulation part[2].

When run we use flags for a disk image, memory size, keyboard layout, and how many cores that the guest should have. The official version used for testing is 1.2.2. Specific compilation instructions are referred to in the appendix.

4.3.3 QEMU-KVM

Since the tested version of Qemu above has not been merged with the KVM branch of Qemu, we will look into the performance of Qemu-KVM. Another reason to test the KVM branch is because of optimizations that have been done on this branch that are not in the main Qemu branch, such as the `--cpu host` flag which expose the host CPU directly to the guest. This flag has become available for newer versions of Qemu that is only available for KVM guests.

When compiling we will not take use of any specific additions, meaning that it will run by its default configuration. This branch of Qemu, will be tested with the same configuration as the regular Qemu tree, with the addition of the CPU host flag and Virtio. The version to be used during testing is 1.2.0.

4.3.4 Virtual Machine Manager and libvirt

Since virtual machine manager is only a graphical layer we test this in conjunction with libvirt. With libvirt being an abstraction layer capable of using both KVM and Xen based guests, both will be tested here. The complete virtual machine manager suite will be compiled from source with no special additions to the configuration.

Libvirt and virt-manager are installed by using available packages in Fedora. The installed versions are Libvirt 0.9.11 and virt-manager 0.9.5.

[1] On multi-user systems this means that the user needs to have root access to install KVM
[2] A test of emulation vs. virtualization in terms of performance should also be trivial.

4.3.5 Xen

For testing the Xen hypervisor and subsequently all Xen/Libvirt based guests, the default package available in the Fedora 17 distribution is used. The choice of using the default package and not compiling from source, is based on the simplicity of installing the Xen package in Fedora. The version used is 4.1.

Xen will be tested through Libvirt using the virt-manager, in addition it will be tested without Libvirt. This will be done with the use of the `xl` command, which is used to handle Xen based guest. To interact with the guest both console and a VNC-viewer will be used.

The guests will be installed in two virtualization modes that are supported by Xen. Firstly we test the most known type of Xen virtualization, paravirtualization (Xen-PV) and lastly we test with Xen hardware virtualization (Xen-HVM). The latter being of course full virtualization. Interestingly enough Xen-HVM takes use of Qemu's device model to handle full virtualization guests, more specifically Xen uses some of the Qemu code to enable device emulation for unmodified guests.

Both Xen-PV and Xen-HVM will be tested both as standalone, using only the tools that come with the Xen distribution, and using Libvirt and the tools that come with Libvirt.

4.3.6 Virtualbox

Since Virtualbox is the most commonplace open-source VMM now available for desktop users we also want to test this. Here we use the rpm package from the Virtualbox website, and the version used will be version 4.2.1. [3] The guests will be installed with the options that are default for VirtualBox, for that reason VirtualBox will differentiate itself from the other virtualization platforms tested by using the VDI disk image format.

4.3.7 Equipment and operating system

To perform the benchmarks I use a lab computer located at the Department of Informatics. This is a HP Compaq Elite 8100 CMT desktop computer. With a Intel Core i7 processor with 4 CPU cores and 8 threads, which gives a computer the impression of having 8 processor cores available. The processor also has three cache levels. Of which level 1 (L1) is 32KB, level 2 (L2) is 256 KB, and level 3 (L3) is 8192KB. The machine is also supplied with 8GB of RAM, and an Intel SSD hard disk of model X-25M with 80GB storage.

For the operating system I have opted to used the Fedora 17 distribution on both the host and guests. Kernel version used has been version 3.6.11 for all installed machines. This kernel version has also been used for `dom0` in Xen.

4.4 Benchmarking suites

This section will talk shortly about each benchmark-suite used in the tests, we will cover some usage of the suites and finally some background for using the suites.

[3] Compilation of Virtualbox is dropped due to complicated compilation, which brings in potentially unnecessary dependencies for the user.

4.4.1 Context Switches[4]

This program is mentioned in the article [26]. This is a simple program to test the time it takes to do a context switch on a Linux system, and will be used for the benchmarking that will be performed.

The application is divided into two programs, the first measures the direct cost of a context switch, while the second measures the indirect cost of a context switch. The direct cost of a context switch can come from several factors, among them saving and restoring registers and reloading of TLB entries. Indirect costs can come from performance degradation from cache sharing and varying costs between workloads with different memory access patterns. Frequent context switching is implemented using pipe communication between two processes. First the direct cost is measured, then the total cost is measured. Lastly the user is required to subtract the direct cost from the indirect cost to get the total cost of a context switch, since the indirect cost includes overhead from the direct cost. Timing of context switching is done using the CPU cycle counter.

The cost of a context switch can also be measured using an access stride of varying size which access data in a nonlinear fashion. This access pattern can greatly affect the cost of a context switch. In addition the application is designed to either use a single core or multiple cores, this means that reconfiguration and recompilation is needed when switching between one and multiple cores.

The application does itself perform three runs for each benchmark and computes the mean, I will perform five runs again which I will then compute the mean of. Testing should be done using sizes that range from 0 to 16384K and different strides in the range 0 to 512B.

4.4.2 Cachebench

The Cachebench benchmark, part of the LLCbench benchmarking suite, is a tool used to benchmark a systems memory subsystem empirically, and specially designed to evaluate the performance of the memory hierarchy of caches. More specifically the benchmark focus is to parameterize the performance of multiple levels of caches present in the processor, in my case level 1 through level 3 caches. Performance measures the row bandwidth in MB/sec.

Since caches on processors are generally small compared to the system memory, i.e. the computer used for my benchmarks has 8GB of memory and a 8MB L3 cache, and since caches are present on almost all available processors, we want to test the performance of the memory subsystem hierarchy. Cachebench is used to establish the peak computation rate given optimal cache reuse. Some of the background for this benchmark to exists is the need for applications with high significance requirements in terms of memory usage and performance, and this will give a basis for performance modeling of systems.

Cachebench uses eight different benchmarks to measure memory and cache performance, namely; Cache read, write and read-modify-write, Hand-tuned read, write and read-modify-write, memset and memcpy. Each of these performs repeated access to data items of varying vector length, and takes timing for each number of items. Of these eight benchmarks I have chosen to use Cache read

[4]This benchmarking suite will be presented as, and mentioned as, Context Switching for the sections were I comment upon this benchmark.

and write. Both are designed to provide us with the read and write bandwidth for varying vector lengths in a computer optimized loop. When the vector size is smaller than the available cache data should come directly from the caches. Sizes tested range from 256 bytes up to 512 megabytes.

4.4.3 LMBench

LMBench is a micro-suite benchmark to test physical hardware and processing units. It is able to test basic memory and file operations, in addition to context switches and signal handling, and is a widely used micro-benchmark to measure important aspects of system performance.

Contained in the benchmarking suite that is LMBench, is a large number of benchmarks that make up the LMBench micro-benchmark suite. These benchmarks are mostly intended to measure system bandwidth and latency, more recent versions also measure instruction-level parallelism. Out of all benchmarks contained in this suite, I have chosen to use two of the benchmarks to measure the memory bandwidth and context switching time, these are known respectively as `bw_mem` and `lat_ctx`.

`bw_mem` measures the memory transfer time for various memory operations. For my testing I have chosen to measure read and write. To run the benchmark the binary for `bw_mem` is run with the appropriate arguments, these are firstly the size to test then the operation, i.e. read or write. The sizes that will be tested will range from 1 kilobyte to 1024 megabytes.

`lat_ctx` is used to measure the time taken for a context switch in a processor. Previous versions measured context switching by using an approach with Unix pipes in a ring, however this approach favored schedulers which only had 1 process running at the time, which is not ideal for multiprocessor architectures. As of newer versions, LMBench3, Context switching is measured in a style similar to LMBench2, with N process rings for each processor. Context switch time is measured between the N rings running in parallel. The results that arise from `lat_ctx` are not totally accurate, the corresponding documentation states that there exists a 10 to 15 percent discrepancy in the reported results, it is recommended to use several runs. For my testing I perform five runs and compute the average, size is given from 0 to 1024 kilobytes and running with 2, 4, 8 and 16 processes. Due to this error margin that is reported, the results are not 100% accurate and should be considered within a 10-15% error margin.

4.4.4 Linpack

Linpack is a benchmark program and collection of Fortran subroutines based on the Basic Linear Algebra Subprogram (BLAS) library. It was designed in the 1970s to measure the performance of then supercomputers. Linpack measures the peak value of floating point computational power, also known as GFLOPS. To test the performance in Linpack we use a program known as HPL, or High-Performance Linpack, which is a portable Linpack benchmarking program to measure the GFLOPS performance on distributed memory computers. We will only run this on a single node both on the host machine and on the virtual machines to be tested.

Configuration of the benchmark is done through a file called 'HPL.dat', this file can be tweaked extensively to get the best results when running this

	1	**2**	**4**	**8**
1x1	4.80	4.44	4.48	4.80
1x2	N/A	9.02	8.71	8.63
1x4	N/A	N/A	15.74	15.75
1x8	N/A	N/A	N/A	13.64
2x2	N/A	N/A	15.40	15.34
2x4	N/A	N/A	N/A	13.81

Table 4.2: Various process grid configurations for HPL benchmark.

benchmark on a given system. As an example, the number of nodes for which the benchmark is to run upon should be taken into consideration when tweaking, as well as the number of processor cores available in total. This is represented in HPL.dat as a process grid, denoted by P and Q. In the configuration used during these tests only one process grid was used, and the size of the process grid was calculated by using the number of cores available, i.e. on a 4-core system P should be set to 2 and Q to 2, or 1 and 4 respectively, the latter is however discouraged due to performance issues. Also, the values of P and Q should always be slightly equal, with Q being slightly larger than P[24]. A small performance measurement of the various P and Q values is given in Table 4.2, problem size was 5000 running with processes ranging from 1 to 8, with different configurations for P and Q where applicable.

The above table was computed only once, no mean was computed, therefore it should not be taken as scientific proof for one process grid configuration. It does show some favor towards having a linear process grid for the HPL benchmark, however convention and the plethora of HPL.dat calculators online will favor the two dimensional process grids whenever possible. With that in mind the process grids used for the experiment was: *1x1*, *1x2*, *2x2*, and *2x4*.

During the setup of the benchmarks, a significant amount of time went into installing HPL. For that reason I have included a short instruction of HPL installation in the appendix.

4.4.5 IOZone

IOZone is a file systems benchmark suite written in C, that tests various workloads and file-system operations to test the I/O performance of a system. Operations performed are, but not limited to, *read*, *write*, *re-read*, *re-write*, of which read and write is used for my testing.

The read benchmark of IOZone measures the performance of reading an existing file, while reread measures the performance of reading a file that has recently been read. Performance of reread is for that reason expected to be somewhat better than the performance of read, due to caching of the recently read file. The write benchmark measures the performance of writing a new file to the system. In this process it is not only the new data that needs to be written, in addition metadata for the newly created file needs to be created and stored, such as directory information and space allocation. When running the rewrite test, which writes to an already existing file, it is to be expected that the performance will be better than from the write operation.

When IOzone is run the user has a plethora of command line options, to

tweak the benchmark for their specific needs or the target they are benchmarking. If not specifying a file size or a minimum/maximum file size, the auto mode chooses file sizes in the range 64 KB to 512 MB. The record length is by default in the range 4 KB to 16 MB.For my testing I have simply chosen to use the auto mode (`-a`), specify the read and write parameters, and specify the file sizes that I will test, record length has used the default values. The testing is mostly inspired by the IOZone test done in[67] which used the same parameters.

4.5 Experiment design

For all of the benchmarks to be performed as fairly as possible, some thought has been put into how all benchmarks should be performed. Mainly with regard to each benchmark suite and the parameters that follow each virtual machine, such as memory size, and how they should be assigned.

For some of the tests the CPU should not necessarily have an impact, however, with the advent of multi core and multi-threaded processors, some of the tests will be performed with 1 and 2 processor cores/threads respectively. This is because I want to see if the systems behaves differently with the presence of more than one core. And how this possibly can affect the results were the number of cores necessarily should not have an impact.

When it comes to sample size, a size of 5 will be used. From all of these samples the mean will then be computed to get the result that will be presented in graphs and tables. We use 5 because it should be sufficient for these tests, and will rule out most timing issues with the processor and give us consistent results.

I will go through each type of experiment, the parameters and what benchmark suite that will be used for the given experiment.

4.5.1 CPU-based tests

The following table shows how the CPU-based tests will be performed. We test three different benchmark suites/programs, use only 2048MB of memory and use a RAW disk image were the operating system uses the Ext4 file system.

CPU-based tests			
Suites	CPU	Memory	I/O
Context switch / LMbench / Linpack	1	2048MB	RAW-image / Ext4
	2		
	4		
	8		

Table 4.3: CPU-based tests

Context Switch Tested with array size from 0 to 16384 bytes[5] and stride from 0 to 512 bytes.

[5]In retrospect it was erroneous to assume that the context switching application took the size input as kilobytes and not bytes as is the case. The results are for that reason missing important data, further explanation follows in Chapter 6.

- `-n #` size to be tested
- `-s #` array access stride

LMBench CTX Used size from 0 to 1024 kilobytes with processes from 1 to 16, increment at the power of 2.

- `-s #` size to be tested
- `#` number of processes

Linpack Uses sizes from 1000, and increments every 2000, up to 15000. Other configurations in the `hpl.dat` file which can be viewed in the appendix.

4.5.2 Memory-based tests

The following table shows how the memory based tests will be performed. For the file-system we use a RAW disk image were the operating system uses the Ext4 file system. We test with Cachebench and LMBench, and use 1 and 2 processors cores for the guest respectively. We test with 2048MB of memory since the benchmarks only focus on memory bandwidth, and not exhaustion of the available memory.

Memory-based tests			
Suites	CPU	Memory	I/O
Cachebench / LMBench	1	2048MB	RAW-image / Ext4
	2		

Table 4.4: Memory-based tests

Cachebench Tested read and write operations, all parameters are default for Cachebench. Parameters are:

- `-d 5` seconds per iteration
- `-e 1` repeat count per iteration
- `-m 29` log2 base of maximum problem size
- `-x 1` number of measurements between powers of 2

LMBench MEM Tested read and write operations, using sizes from 1 kilobyte up to 1024 megabytes. Parameters are:

- `#` size
- `rd` or `wr`, read or write

4.5.3 I/O-based tests

The following table shows how the file-based tests will be performed and how the various parameters are to be used. Here IOZone is used to test the file-system performance. All guests are tested with 2048MB of memory with 1 and 2 cores respectively. For the file-system a RAW-image is used, a Qemu Qcow2 image and lastly a logical LVM partition is used. While not listed in the below table VirtualBox will utilize the VDI disk image format, which will be compared to the RAW results.

File-based tests			
Suites	CPU	Memory	I/O
IOZone	1	2048MB	RAW-image
			Qcow-2
			LVM
	2		RAW-image
			Qcow-2
			LVM

Table 4.5: File-based tests

IOZone Tests read and write using file sizes 64, 128 and 256 megabytes respectively. Parameters are:

- `-a` auto mode
- `-i 0` test read and write
- `-s #` size

4.5.4 Platform configurations

Here I will briefly describe the various configurations for the different virtualization systems that has been tested and benchmarked. Most importantly I will list the abbreviations that has been used for the various virtualization suites during testing, and that will appear in figures to come.

QEMU and KVM

For Qemu and KVM there has been two variations of Qemu in use, the main branch of Qemu, namely Qemu version 1.2.2, and the KVM branch of Qemu, namely `qemu-kvm` version 1.2.0. During testing the main branch of Qemu has only used one configuration, while the `qemu-kvm` has used three different configurations with minor differences to test various configurations that has been said to increase performance. However the enable KVM flag in `qemu-kvm` has not been used as this should be mute when using this branch.

In the below figure I list the abbreviations used for Qemu and KVM and briefly describe what is special about that configuration.

`qemu-1.2.2-ekvm`	Qemu main branch with enable kvm flag in addition to default options.
`qemu-kvm`	`qemu-kvm` with default options and no special configuration.
`qemu-kvm-host`	`qemu-kvm` with the cpu flag as *host* to expose the hosts cpu onto the guest.
`qemu-kvm-host-virtio`	`qemu-kvm` as above with cpu host flag, in addition the *virtio* driver has been enabled for the disk image.

Figure 4.1: The different QEMU configurations and abbreviations.

Xen

For Xen there has been two main configurations in use, namely Xen with full virtualization and Xen with paravirtualization. All configuration have been done through the `xl.cfg` files, using the *Xl* tool-stack in favor of both *Xm* and and *xend*, mainly since *Xl* has been encouraged for use in version 4.1 and enabled by default as of Xen version 4.2[63]. In addition, for Xen with full virtualization and LVM disk, I have tested with the so called *PV on HVM drivers* (*PVHVM*). These drivers have been enabled using the `xen_platform_pci` option in the configuration files, in addition this was tested with two different disk configurations as demonstrated below.

`xen-pv`	Xen with paravirtualized drivers, disk configured as xvda.
`xen-hvm`	Xen with full virtualization, disk configured as xvda.
`xen-hvm-pci`	Xen full virtualization with `xen_platform_pci=1` and disk as xvda.
`xen-hvm-pci-hda`	Same as `xen-hvm-pci` with disk configured as hda instead.

Figure 4.2: Xen configurations and abbreviations.

Libvirt

Libvirt was used to test both KVM and Xen with both full virtualization and paravirtualization. For all three of these suites no special configuration was done, they all used the default options that were enabled by the installer, with the exception of memory and number of cores that was adjusted to the various benchmarks and options being tested.

`libvirt-kvm`	KVM enabled libvirt.
`libvirt-xen-hvm`	Xen enabled libvirt with full virtualization.
`libvirt-xen-pv`	Xen enabled libvirt with paravirtualization.

Figure 4.3: Libvirt configurations and abbreviations

Virtualbox

For Virtualbox all default options were used, with of course the different configurations with regard to number of cpu cores available. The main difference in terms of configurations with virtualbox as opposed to the KVM-based and Xen-based suites, is that Virtualbox used its own disk image format, *vdi*, as this was enabled by default and a *raw* disk image as used in all the other suites was possible to use.

`virtualbox`	Virtualbox with default options.

Figure 4.4: Virtualbox configuration and abbreviation.

Chapter 5

Results

5.1 Introduction

This chapter will present the results from the benchmarking that has been undertaken in this thesis. The results will be presented in the following order:

CPU-based benchmarks

High Performance Linpack

LMBench Context Switching

Context Switching[26]

Memory-based benchmarks

Cachebench

LMBench Memory Bandwidth

File-systems benchmark

IOZone

Each graph and figure will have comments and a comment for that benchmarking suite in its entirety. A minor conclusion for each sub-test will follow in the last section of this chapter. A complete conclusion for this thesis and all benchmarks will follow in the next chapter. Note that all tests have been performed five times and a mean is then computed from these results, for that reason some deviation may occur in these results and they most certainly will differ with results from other hardware, operating systems or varying versions of the benchmarks and operating systems. All tests have been run on the same computer with the host running Fedora 17 3.6.11 kernel and the same operating system for the guests with a minimal installation.

The conclusion with regard to these results will be presented in the next chapter, and will comment upon my results with regard to the articles presented in Section 4.2.

5.1.1 Regarding the Host benchmarks

The benchmarks that have been run on the host as a baseline for comparison, have when possible, been run with the option of using only one core, i.e. Context Switch and High Performance Linpack. For those where this is not an option, i.e. Cachebench and IOZone, they have been run one time for 1 core and one time for 2 cores, however all cores on the host has been present and available. For that reason, the host results for 1 and 2 cores have been run with the same parameters. Note that the HPL benchmark has been run in such a way that it only uses the number of cores that it is asked to use, 1, 2, 4 and 8 in this case.

5.2 CPU-based benchmarks

This section will present the results from the CPU-based benchmarks. I will start with the HPL-benchmark, then proceed with LMBench Context Switching (CTX) and finally Context Switching.

5.2.1 High Performance Linpack

Here the results from the High Performance Linpack (HPL) benchmark will be presented. We will start by commentating upon the various configuration and overall results before going more into detail of each processor configuration, then we will arrive at a minor conclusion regarding the HPL benchmark.

As already mentioned in a previous section commenting upon the HPL benchmark, this benchmark has been performed with process grids that represents the number of available cpus and cores. The results presented reflects the above comparison of process grids with regards to performance within a process grid of a given size. I.e. the benchmark run with only one cpu, reflects this by never peaking above 5 Gflops. The results also presents a substantial difference between several leading virtualization platforms, most notably is the difference between KVM-based virtualization suites against Xen-based suites.

Note in the histograms that the host is represented by a line to give an impression of how close to the host system the virtualized systems are in terms of performance.

CPU 1 As can be seen in the figure 5.1, none of the virtualization suites pass 5 Gflops in performance, however they are all quite similar, all staying within 1 Gflop of each-other. With the size set to 15000, the lowest performance is given by *xen-hvm* at 3.87 Gflops, and the highest being qemu-kvm at 4.72 Gflops. Interestingly the host achieves a performance of 4.61. Letting, among others, qemu-kvm to pass the host slightly in performance, this is caused by small deviations in the data since we are working with mean values.

CPU 2 With 2 cpu cores enabled in the guests, the various platforms begin to stand out in front of each-other, as we can see in figure 5.2. Most notably is qemu-kvm and the host, as they both are quite similar. While Libvirt-xen with paravirtualization is quite in the opposite end of the scale. At 15000 the host achieves 8.72 Gflops, while libvirt-xen-pv achieves 6.05 Gflops in performance.

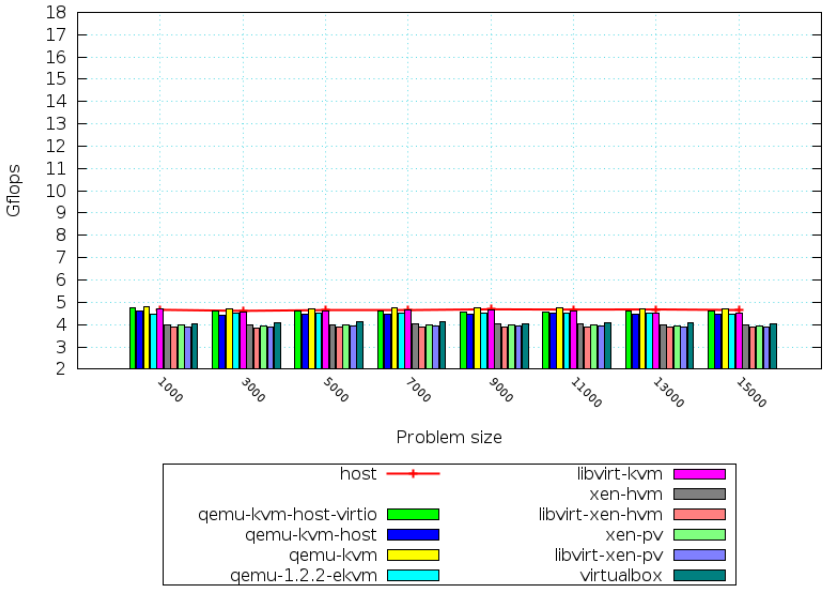

Figure 5.1: HPL benchmark for 1 cpu core.

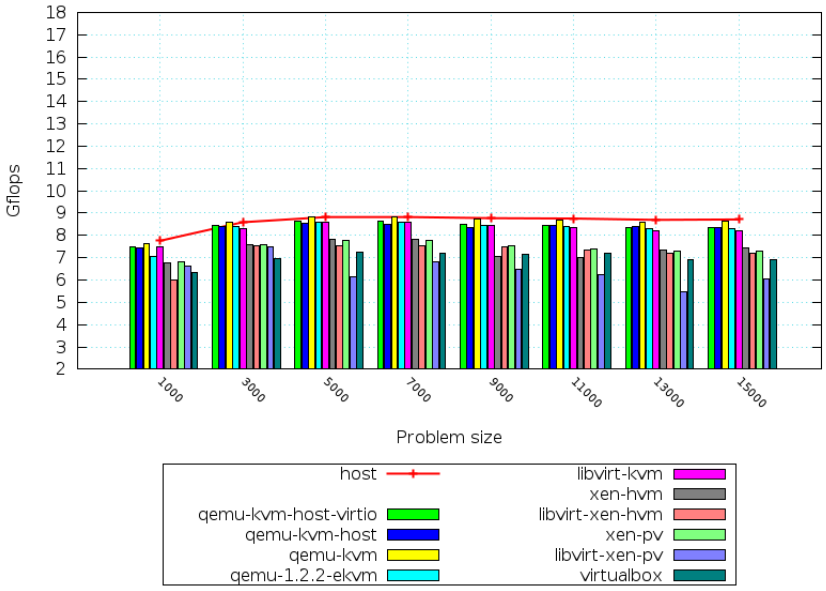

Figure 5.2: HPL benchmark for 2 cpu cores.

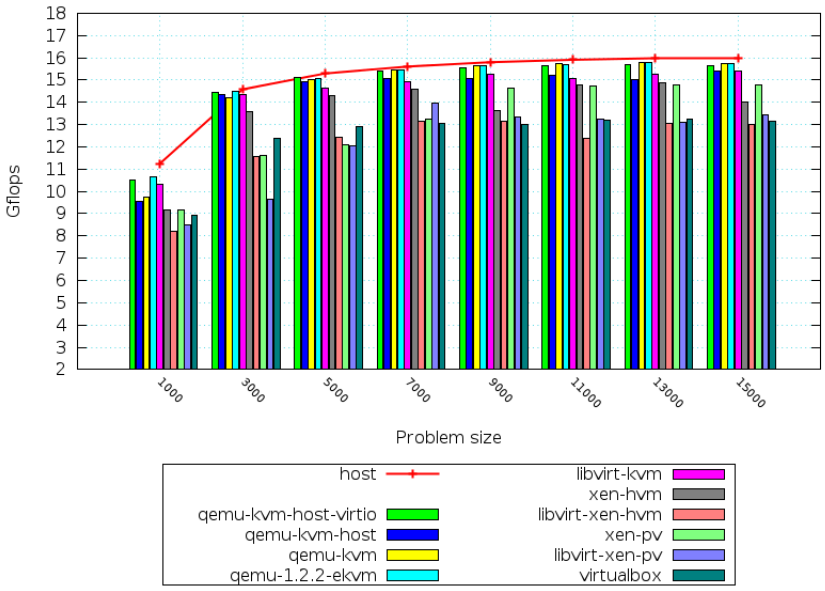

Figure 5.3: HPL benchmark for 4 cpu cores.

This is quite a substantial difference, while libvirt-xen-pv struggles to pass 6 Gflops, its cousin, xen-pv finds itself closer to 7 Gflops. What causes the difference between the two, is most likely configuration-wise. Both have been installed minimally, xen-pv using a minimum of settings in its xl.cfg file, and libvirt-xen-pv have been installed using virt-managers graphical user interface with only the disk image, memory, cpu cores, and paravirtualization enabled, all other settings are default. What causes the difference could in some part be due to overhead using libvirt. What causes the difference between libvirt-xen-pv and qemu-kvm, and the KVM based suites in general, is definitely a part of the way KVM and Xen handles transitions between guest and host, and privileged instructions.

CPU 4 The results in figure 5.3 have a maximum peak value at just above 15 Gflops, also the results flatten out from sizes above 5000, from which some virtualization suites stand out in terms of performance. The host and the KVM based suites all hover around the 15 Gflops mark, while libvirt-xen-hvm hovering around 13 Gflops along with libvirt-xen-pv and virtualbox.

The pattern that we saw for 2 cores, repeats itself here. libvirt-xen-pv is outperformed by the KVM based suites, however this time, libvirt-xen-hvm performs worse than libvirt-xen-pv. However the two are still fairly close to each other, as well is virtualbox hovering at the same performance level as the two. This still raises the question about the way Xen based suites is handling privileged instructions, against the way KVM handles the same instructions.

CPU 8 With 8 cores enabled, see figure 5.4, the same pattern that we saw with 2 and 4 cores repeats itself. However the performance does not peak above 15 Gflops, this is mainly because we are only using 4 physical cores. We are

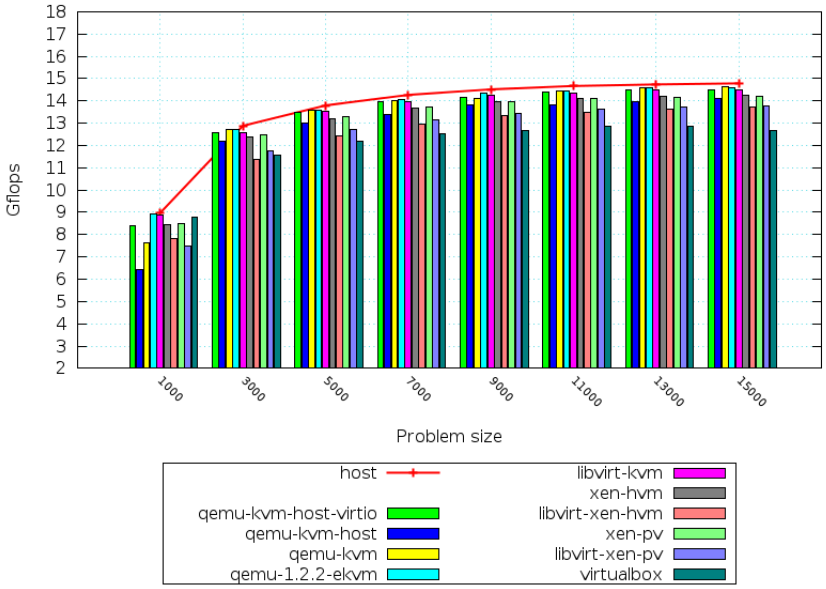

Figure 5.4: HPL benchmark for 8 cpu cores.

actually using 8 threads thanks to Intel's Hyper-Threading technology. That is also one of the reasons why all of suites are closer to each other than they have been in the two previous tests.

The best performing suite is yet again KVM based, namely qemu-kvm. While this time the worst performing suite is Virtualbox. All of the Xen based suites are closer together than they have been for the two previous tests, xen-hvm and xen-pv are about the same, while their libvirt cousins performs slightly worse than the two.

Comments The interesting part of these results is how close certain virtualization suites, most notably the KVM-based, are to the host in terms of performance, in some cases they even pass the host in performance. However, the most noteworthy is how different the KVM-based and the Xen-based suites are. As an example, the benchmark run for 4 available cpu cores highlight this difference, the Libvirt with Xen full virtualization (libvirt-xen-hvm) achieves about 13 Gflops, while qemu-kvm and qemu-1.2.2 with KVM enabled achieves above 15 Gflops in performance.

This difference is also reflected in all the results that this presented from this benchmark. The KVM-based virtualization suites all achieve better performance in Gflops than the Xen-based suites as well as Virtualbox. The results here also comment upon the Hyper-Threading technology that is present in the host CPU. Which gives the operating system the impression of having 8 available CPU cores, while they are in reality threads. This also explains the slight performance drop when increasing the process grid in size from 4 to 8. The performance is however substantially faster with "8" cores than with 2 cores.

What causes these differences between the various virtualization suites is most likely how they handle critical regions and transitions. Differences between

the various KVM based are all configuration based, as all run on the same version of KVM and the same host kernel, however they all have slight differences. As and example, the qemu-kvm-host suite uses a parameter to qemu to directly expose the host cpu to the guest, however qemu-kvm exposes a generic cpu onto the guest (called qemu64, or in some cases kvm64) which to some extent makes the guest aware of being virtualized.

5.2.2 LMBench Context Switch (CTX)

This section will present the results for LMBench Context Switching (CTX). The tests have been run with sizes ranging from 0 to 1024K, and with a number of processes from 2 to 16. The results will be presented with the time in micro second units on the Y-axis and the size on the X-axis. The results from the host and Virtualbox are presented just below their KVM and Xen counterparts, Virtualbox is presented along the rest when using 1 core as well. This is done because the results from the host and virtualbox differentiates themselves in such a manner that the results are best presented for themselves, and also to keep the range of the Y axis in a more readable region. For that reason the reader should note the range on the Y-axis for easier understanding of the graphs. The graphs will be presented with 2 processes first, before we move on to 4, 8 and 16 processes.

Note that the host has been tested with all cores available for the four runs that are tested. The results presented here represent bare metal performance of the host with a default system configuration.

2-processes When we examine the results with 2 processes we can see a pattern that emerges for the results with 1 and 2 processor cores. Both show that the paravirtualized Xen suites and Libvirt with fully virtualized Xen uses more time to make a context switch than the KVM based suites, as well as fully virtualized Xen without Libvirt. Virtualbox with 1 core performs the same as the KVM based, the Host is the one that takes the most time to perform a context switch. With 1 core the host takes the most, while with 2 cores, Virtualbox uses substantially more time time to perform a context switch.

Looking at the results with 4 and 8 cores, we can see that the tables have turned a bit. This time it is the KVM based suites that takes longer time, while the Xen based suites are more consistent in their performance. Virtualbox takes the longest time to switch with 4 and 8 cores.

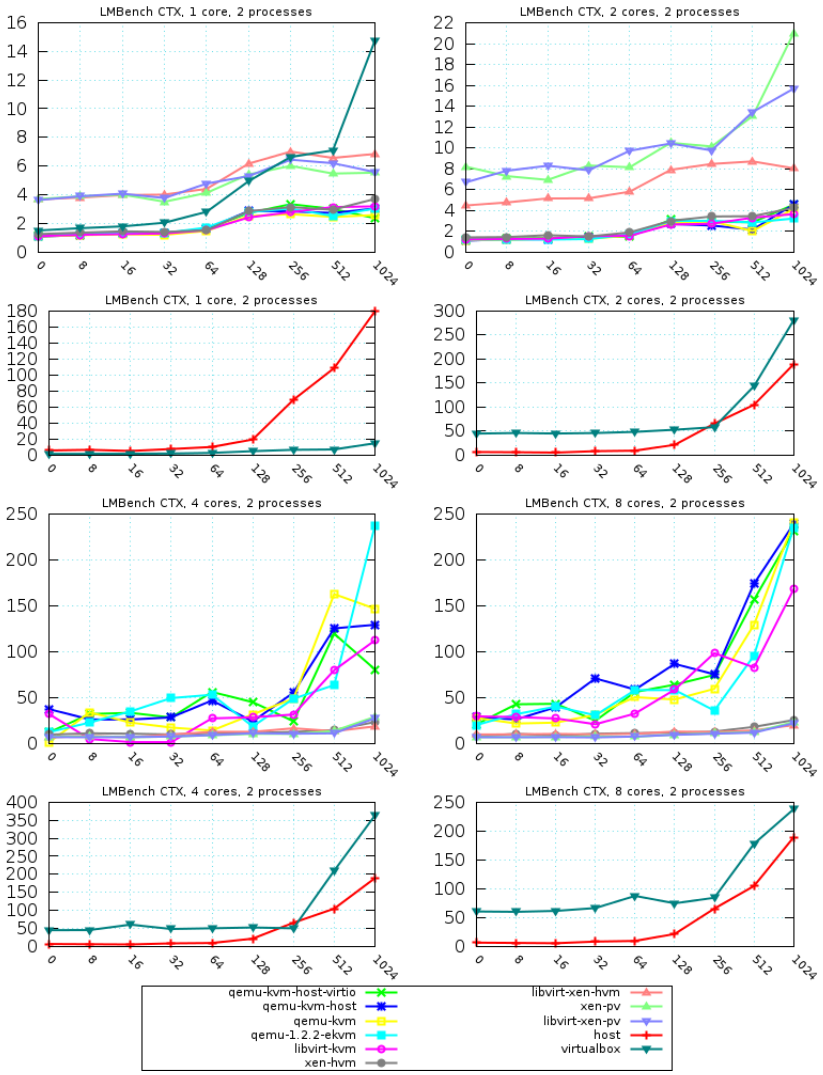

Figure 5.5: LMBench CTX with 2 processes.

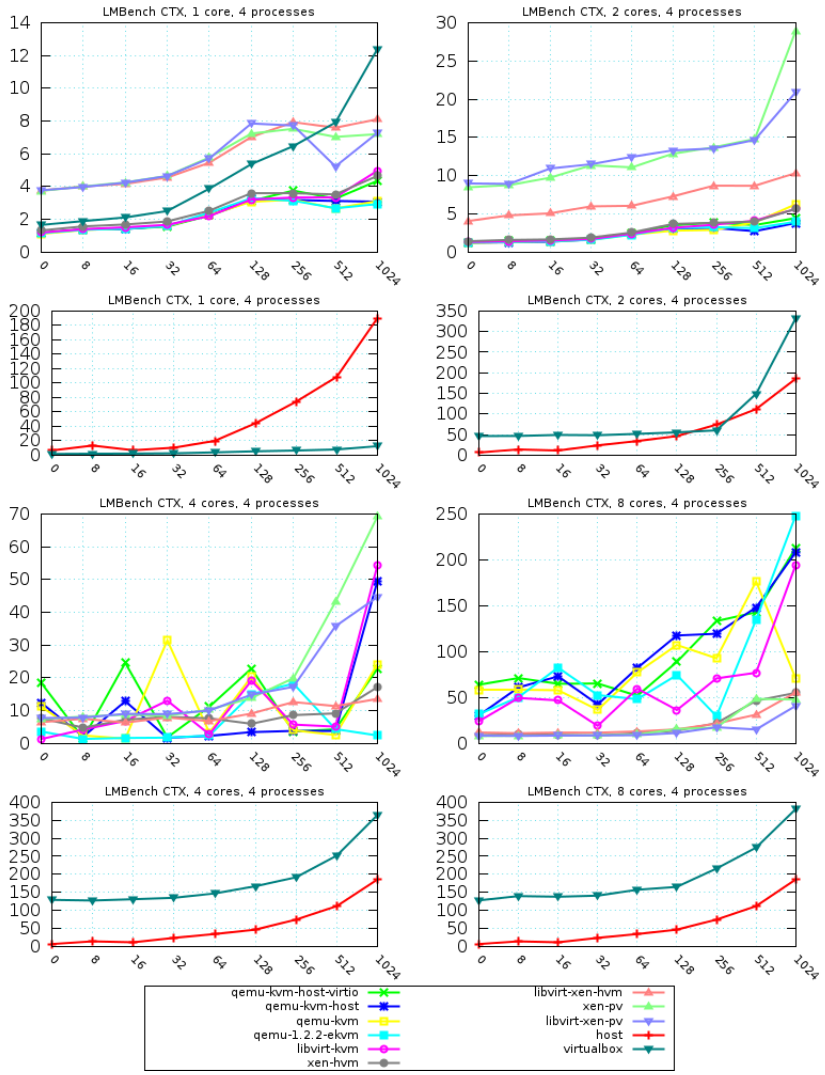

Figure 5.6: LMBench CTX with 4 processes.

4-processes When we increase the number of processes to 4, we initially do not see that much difference from the previous results with 1 and 2 cores. With 4 and 8 cores, we see that the fully virtualized Xen suites are more consistent in their context switch time. Their paravirtualized cousins using more time with 4 cores when the size increases. The KVM based suites are more erratic in their context switch time, with both 4 and 8 cores. Interestingly all KVM suites increase their context switch time as the number of present cores increases. Virtualbox performs the same as the KVM based suites with 1 core present, however the performance drops when the number of cores increases.

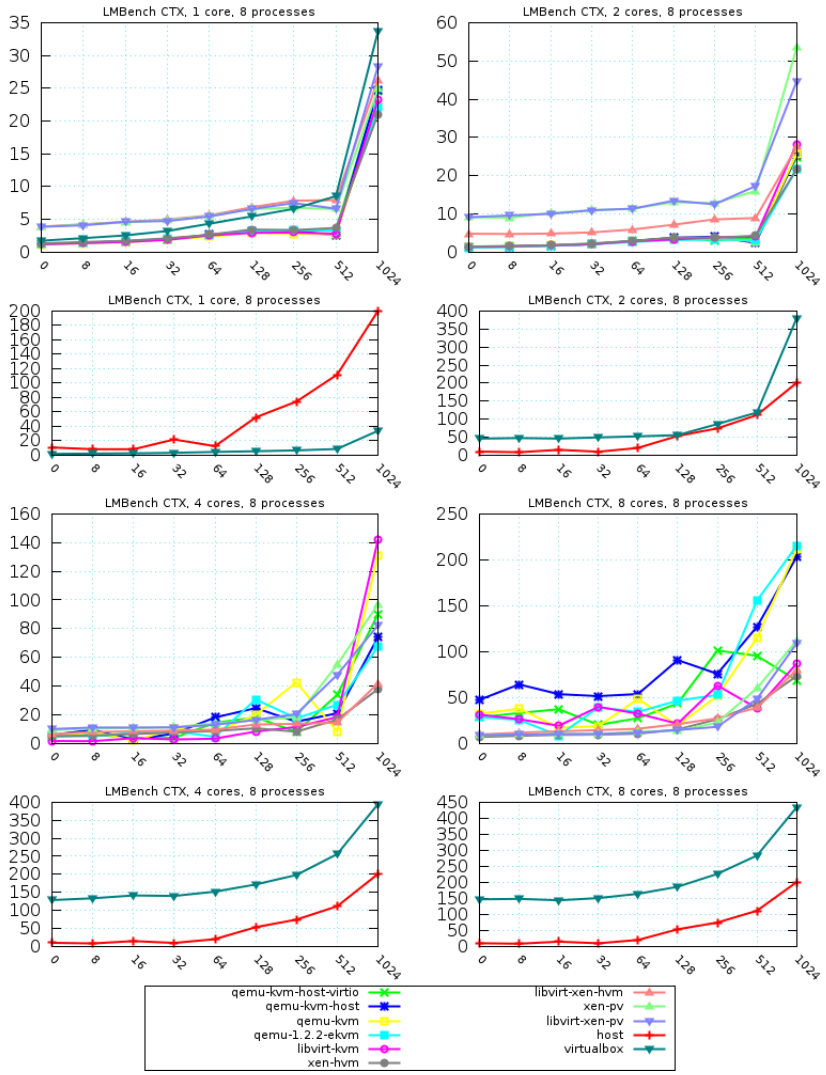

Figure 5.7: LMBench CTX with 8 processes.

8-processes With 8 processes we have the three Xen based suites, xen-pv, libvirt-xen-pv and libvirt-xen-hvm taking some longer to perform a context switch as opposed to xen-hvm and the KVM based suites with 1 and 2 cores. With 4 and 8 cores the fully virtualized Xen suites deliver the most stable performance, the paravirtualized behaving in the same manner as with 4 processes. Virtualbox performing almost the same as the KVM based suites with 1 core. While all suites perform better than the host. Increasing the number of cores, Virtualbox again begins to drop in performance. For the KVM based suites we also see again that performance is lowered, and the results are more erratic.

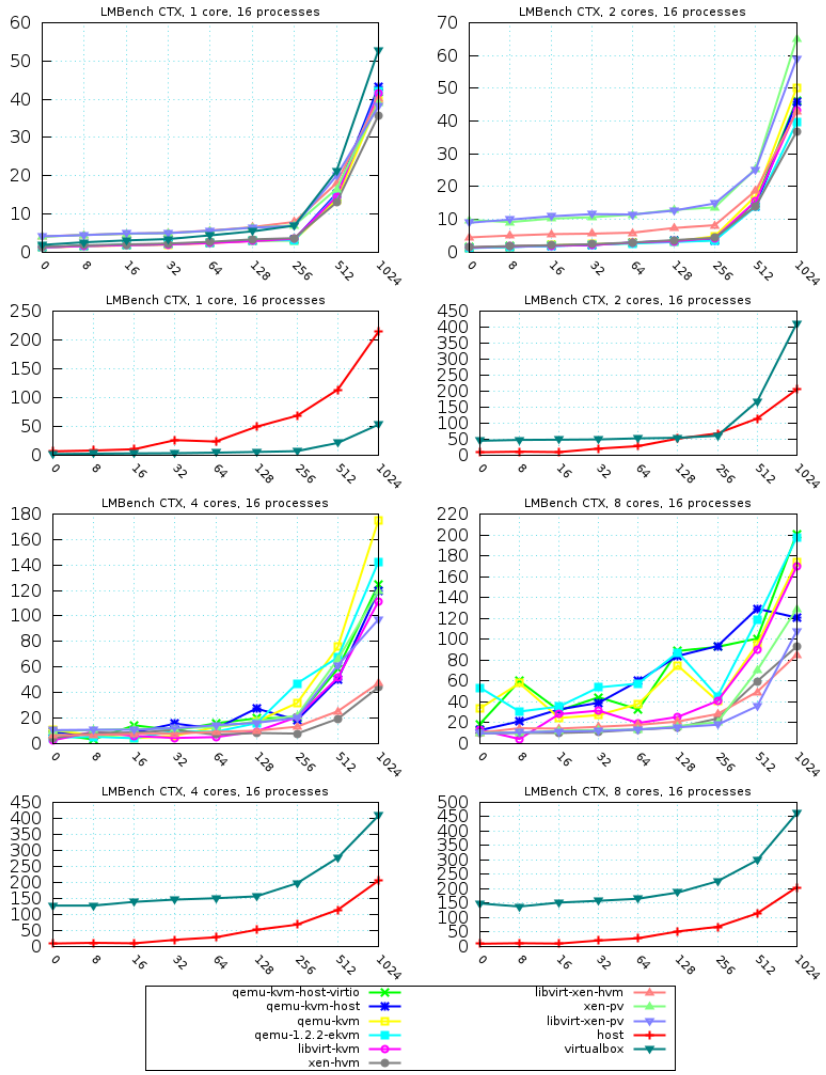

Figure 5.8: LMBench CTX with 16 processes.

16-processes When we increase the number of processes to 16, we almost see the same results as we did with 8 processes. There is still some difference between the KVM based suites and some of the Xen suites. Virtualbox performing the same with 1 core, however performance drops substantially when the number of cores increase.

Comments

In the above results we have seen that there is surprisingly little difference between the KVM based suites in context switch performance, even the fully virtualized Xen configuration, xen-hvm, performs the same. The paravirtualized Xen suites, as well as fully virtualized Xen with Libvirt, libvirt-xen-hvm, have a performance that is at least 3ms higher with 1 or 2 cores. Virtualbox on the

other hand performs the same with 1 core present, wile with 2 cores and up, i.e. performance drops from 1.5 to 44ms with 2 processes and size 0.

As is understandable, the overall performance of the benchmarked suites, as well on the host, has dropped when the size has been increased and when the number of processes increased. I.e. qemu-kvm has a performance of 1.2ms with size 0 and 16 processes, while this is 39ms when the size is increased to 1024K. This is quite understandable since the cache is filled with the data that is used between the 16 processes. What becomes interesting about these results is that performance has a substantial drop when the number of cores is increased for the guests, with regard to certain virtualization suites.

If we are to consider which suite has delivered the best performance, regardless of the performance of the host, and possible shortcomings of the benchmark. The results are then twofold. When the benchmark is run on a guest with 1 and 2 cores, regardless of the number of processes, the KVM based suites performs the best. However, when the number of cores is increased to 4 and 8, the KVM based suites starts to 'misbehave'. I.e. the results are more spread, and seemingly more erratic. We can see that all the Xen based suites deliver consistent results and have a better performance with many cores present in the guests. A possible explanation could be based on the architecture of the hypervisors. Where KVM gets scheduled alongside user-space processes, `dom0` and `domU` of Xen gets scheduled alongside each-other in terms of VMs. Making user-space processes in `dom0` less of an issues as it is for KVM. Further allowing the increase of cores to be less of an issue with regard to other processes that use the CPU.

5.2.3 Context Switching

This section will focus on the results from the Context Switching program presented in[26]. For these experiments I have used a size that ranges from 0 to 16384B and a access stride from 0 to 512B, size and stride increase by the power of 2. The results will also be presented in stride ranging from 0 to 512 with all cpu configurations presented at the same time.

The reason for the size to be given in bytes and not in kilobytes, is because of an error on my part, mistakenly assuming that the size parameter was given in kilobytes, and realizing this too late. The reason that this matters is because the tested array size never exceeds the size of the system cache. While this is crucial for the results to be as accurate and relevant as possible, there are still some interesting results that have risen from this benchmark.

The results will first be presented with sizes 0 and 16384 bytes, then I will present the results with access stride 0 and 512 bytes.

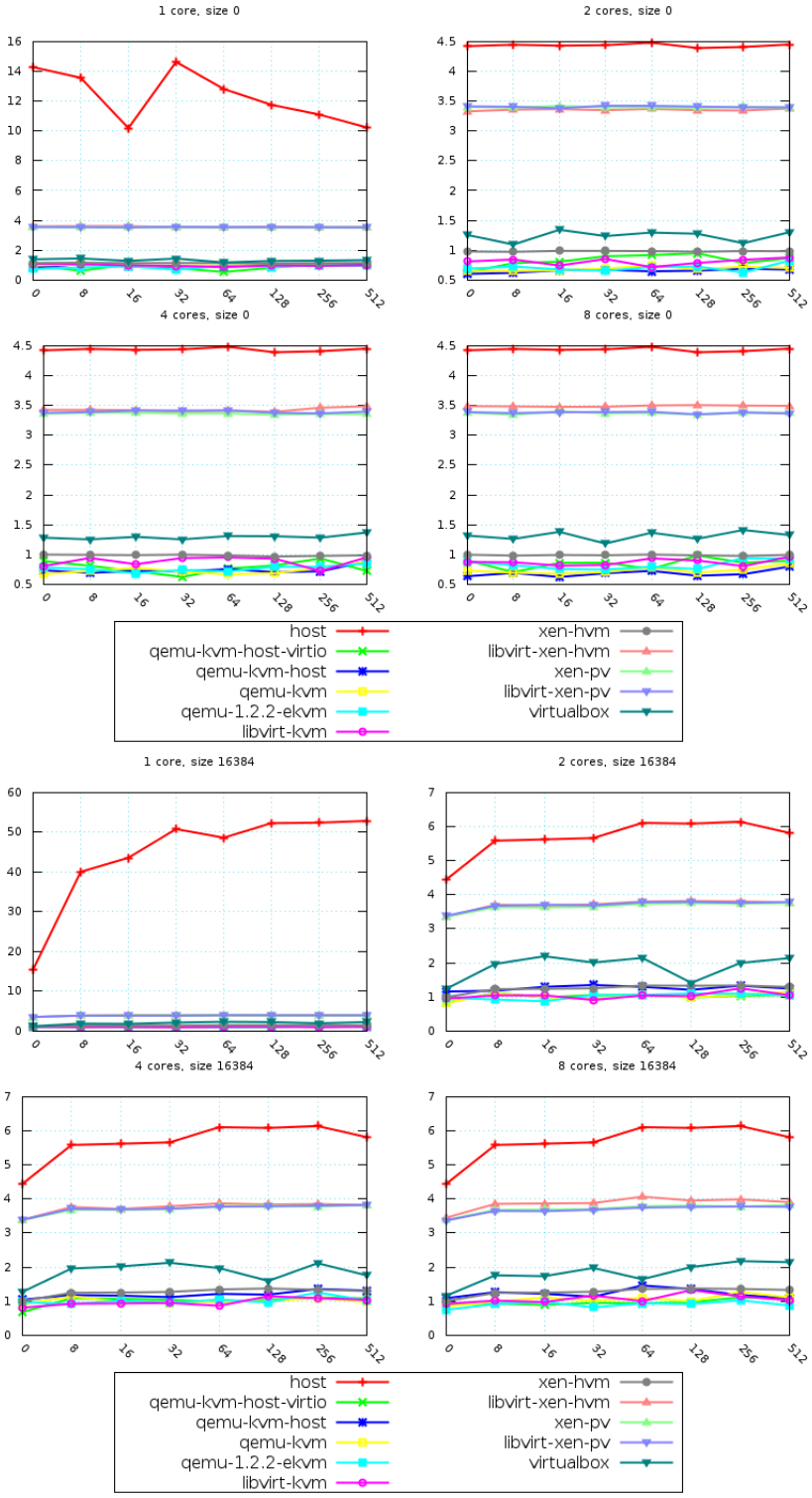

Figure 5.9: Context Switching with size 0 and 16384 bytes.

From the results with constant size and varying access strides in Figure 5.9 we can see that the performance is more or less constant for the tested virtualization suites, of course caused by the sizes being smaller than supposed. What is interesting about these results, as we saw with the LMBench CTX results, there are three Xen suites that have a slightly lower performance than the KVM based suites and Xen with full virtualization. The three suites in question are both Xen configurations with paravirtualization (PV) and Xen HVM with Libvirt. The KVM based suites, with xen-hvm, has a performance that is located around 1ms, while the three mentioned Xen suites have a performance around 4ms. Virtualbox delivers performance which can be found between the KVM based suites and the three Xen based suites, at approximately 1.5-2ms.

What is really interesting about these results, regardless of the size being tested, is the fact that all virtualization suites that have been tested have performed better than the host in performing context switches. A possible reason reason for this to be the case is because of optimizations in the guests, i.e. code in the Linux kernel, which do not cause exits to the hypervisor when switching, as well as the existence of extended page tables in the host processor.

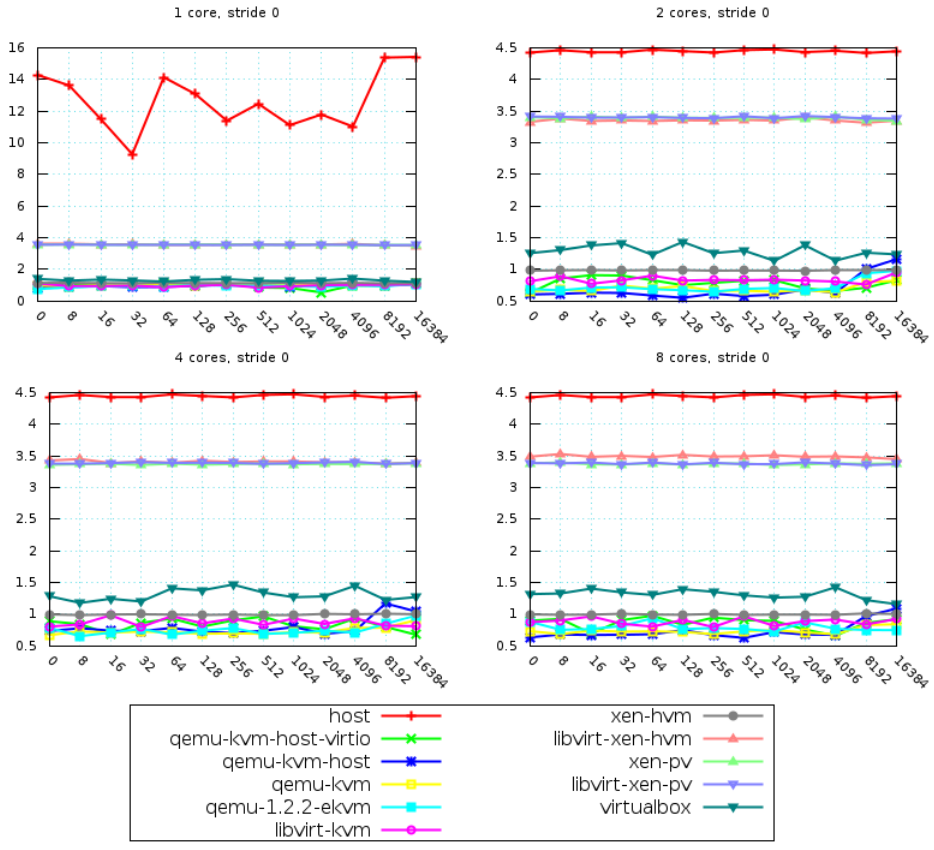

Figure 5.10: Context Switching with stride 0.

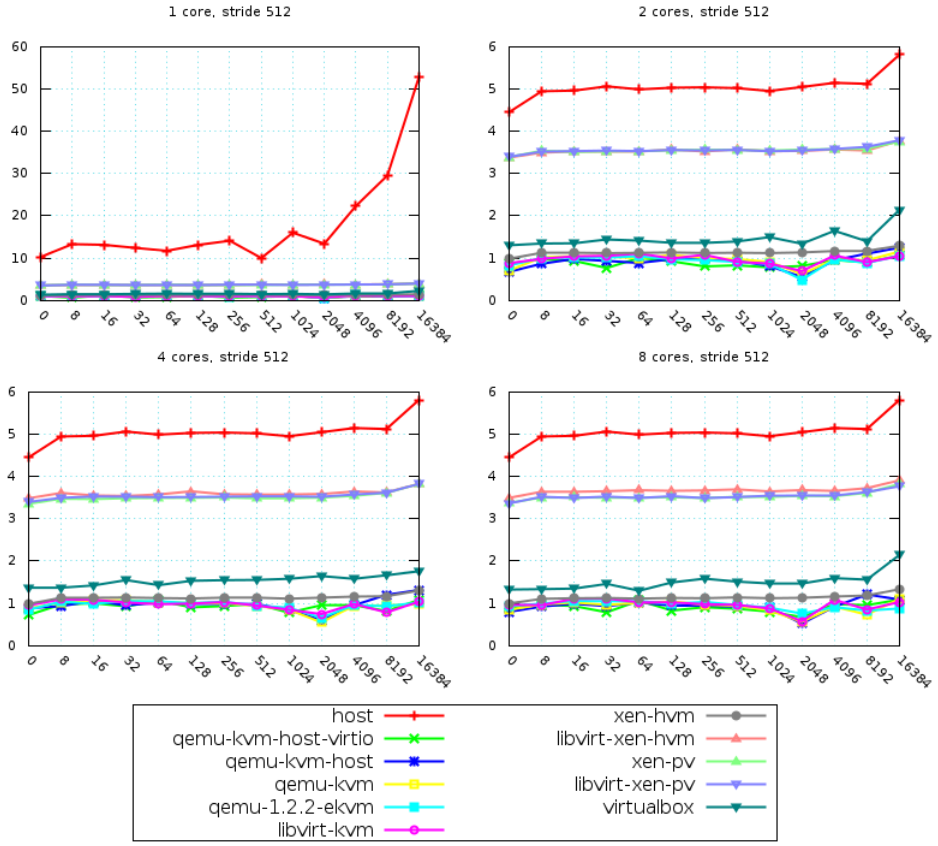

Figure 5.11: Context Switching with stride 512.

When we look at the results with a constant access stride and increasing array size, figures 5.10 and 5.11, we can see the same results we saw before. The KVM based suites with xen-hvm is performing the best, taking the least time to perform a context switch. While the three Xen suites mentioned before, have a degradation in performance as opposed to KVM and xen-hvm. Virtualbox does deliver performance that is similar to the best performers for all configurations. The host does still deliver a performance that is slower than what we see from the virtualized suites.

Comments

With a grave error on my part, the tested sizes for context switches has been far too low to yield as complete results has I had hoped for. However the results presented does highlight some difference between various virtualization suites. The best performing virtualization suites being all of the KVM based suites, all delivering context switch times around 1ms. Alongside the KVM based suites we find Xen with full virtualization, xen-hvm, that delivers more or less equal performance to the KVM based suites. Virtualbox also delivers good performance in context switches for all processor core configurations, close to the KVM based suites. There are also three Xen based suites that have delivered performance that is about 2ms higher than the other tested suites.

Here we find Xen-pv, libvirt-xen-pv, and surprisingly libvirt-xen-hvm, the fully virtualized Xen suite with Libvirt.

What becomes interesting about the presented results, is that it is the host that the delivers the worst performance, which is interesting because this is bare-metal performance. An explanation for this behavior is most likely due to the existence of page-table optimizations in hardware.

5.2.4 Comments to the CPU benchmarks

From the results that have been gathered and has focused on CPU performance in terms of Gflops and context switch time we have, albeit some of the context switch results could have been more thorough, established some impression of the performance present on the benchmarked virtualization suites. We saw in the HPL benchmarks that the qemu-kvm was among the top performers closely followed by other KVM based suites. Of the Xen based suites we saw the those that not use Libvirt, xen-hvm and xen-pv, performed the best of the Xen suites. Of all the benchmarked suites we saw that Virtualbox had the lowest performance, while none of the tested suites surpassed the host. When testing with 4 cores present, the peak performance was reached. With 8 cores enabled in the guests we saw a slight performance degradation, highlighting the presence of Hyper Threading technology on the host. It would be interesting to see how much the performance would have dropped if a guest were to have 16 cores enabled, or if we had several guest present using more than the present number of cores.

Moving on to the two benchmarks that measured the context switch time. We firstly looked at the results from LMBench which suggested that KVM based suites performed the best along with the fully virtualized xen-hvm. The three remaining Xen suites delivered context switch times that were a bit slower. Virtualbox Interestingly delivered performance that were equal to the KVM based suites with 1 core, when the number of cores where increased we saw a huge degradation in performance. The host also saw performance that were slower than the virtualized guests. When the number of cores was increased we saw a degradation in performance for the KVM-based suites, while the Xen-based suites continued to perform consistently. This behavior is most likely attributed to by the architectural differences between KVM and Xen with regard to scheduling of the VMs and user-space processes.

Secondly if we are to consider the results from the Context Switch benchmarks, we can see that the results are comparable to the LMBench results with 1 and 2 cores. If we are to compare the performance with 4 and 8 cores from LMBench, the results are quite different. This is most likely due to the tested size being to low as opposed to LMBench. However all the results highlight the same findings where comparable, with KVM having slightly better performance than Xen. Virtualbox does however not have the same performance degradation when the number of processor cores is increased as it did when tested with LMBench.

Interestingly the host results suggest that the host takes the longest to perform a context switch. While this is quite interesting, the cause of this difference in performance can possibly be attributed to the hardware virtualization additions. Specifically the additions to enable hardware assisted paging, known as Intel EPT and AMD RVI. Findings from VMWare[5] suggested that a 48% in-

crease in performance for MMU-intensive benchmarks and a 600% increase in performance of MMU-intensive microbenchmarks is possible with these additions. Largely explaining why the host seems to take the longest to perform a context switch, as well as attributing to how far virtualization technology has come since Che et al[10] tested LMBench context switching on both KVM and Xen. The usage of EPT and RVI in virtualization suites is for that reason quite obvious to achieve the best performance.

5.3 Memory-based benchmarks

This section will cover the memory-based benchmarks, firstly we view the results for Cachebench, and then from LMBench. Lastly I will comment upon both benchmark results. The conclusion with all benchmarks in mind can be found in the next chapter.

5.3.1 Cachebench

This section will focus on the Cachebench benchmark suite. The tests have focused on measuring the read and write speed of the virtualized systems. First I present the results from the read test with 1 and 2 cpu cores, and then the results for the write tests with 1 and 2 cpu cores. All figures are presented with the time in MB/sec on the Y-axis and an increasing size from 256-bytes to 512-megabytes on the X-axis.

Read with 1 core For the read benchmark with only one core available, in figure 5.12, a clear pattern emerges as to which suites that has the best performance. All the KVM based suites are fairly close to the host, while the Xen based suites find themselves below KVM, about 400 MB/sec. VirtualBox on the other hand finds itself between the two.

All suites behave linearly, meaning that there are no sudden drops in performance . They all hover around the same speed for all sizes, only varying about 50 MB/sec. The Xen based suites are more consistent than the host, KVM based and virtualbox, with minor variations in performance. For that reason, the only drop in performance for the Xen based suites can be clearly seen to be around the 8 MB size, which is also the L3 cache size. This can be seen for all the other suites as well, however it is not as clear for those as it is for the Xen based.

Read with 2 cores With two cores enabled there is no significant difference between this and with one core, see figure 5.13. The KVM based suites have to some extent surpassed the host, which is most likely caused by the presence of another core as the host has been run with all cores available both times. Again the Xen based find themselves around 2200 MB/sec and Virtualbox hovering between the Xen based and KVM based. The same pattern we saw with one core, where the speed takes a minor drop when the size becomes greater than the L3 cache size, is visible here as well.

Write with 1 core For the Cache write benchmark the results are more interesting than for the read benchmark, see figure 5.14. The first thing about

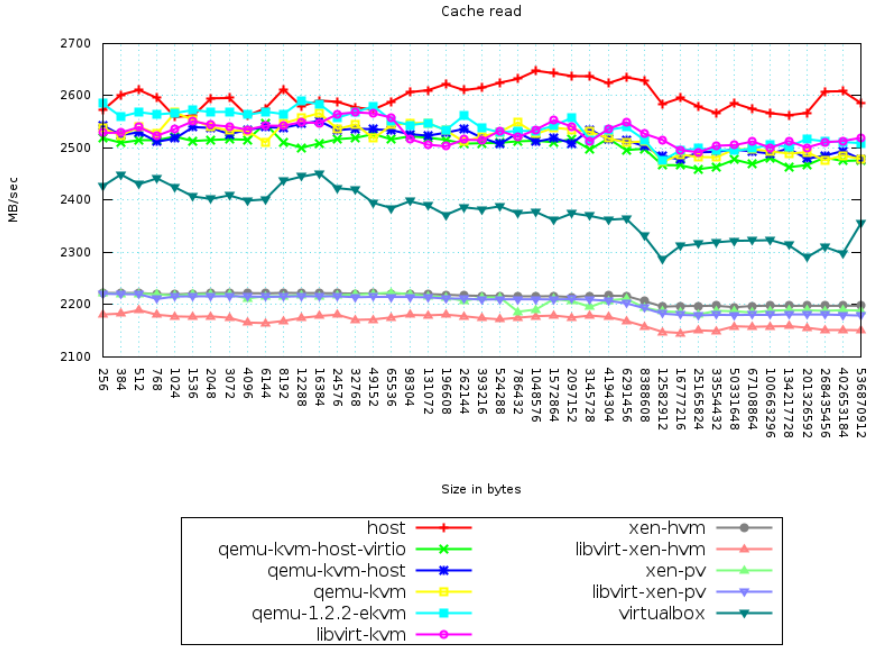

Figure 5.12: Read with 1 cpu core.

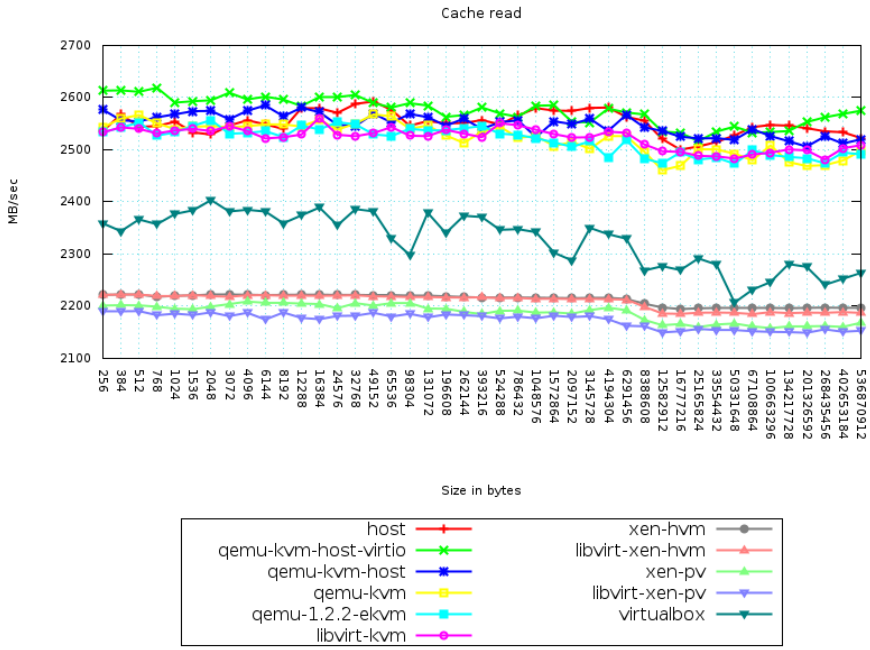

Figure 5.13: Read with 2 cpu cores.

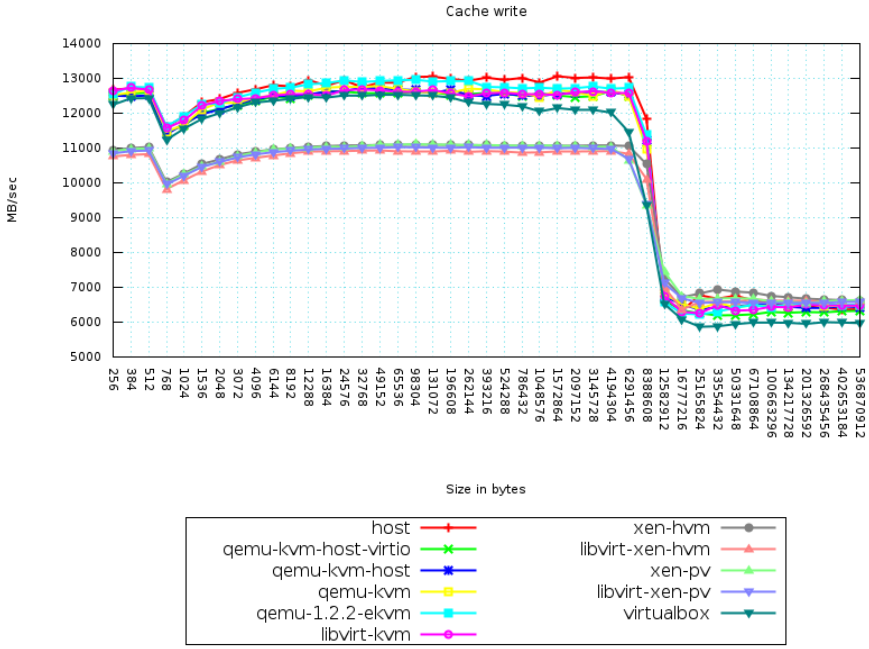

Figure 5.14: Write with 1 cpu core.

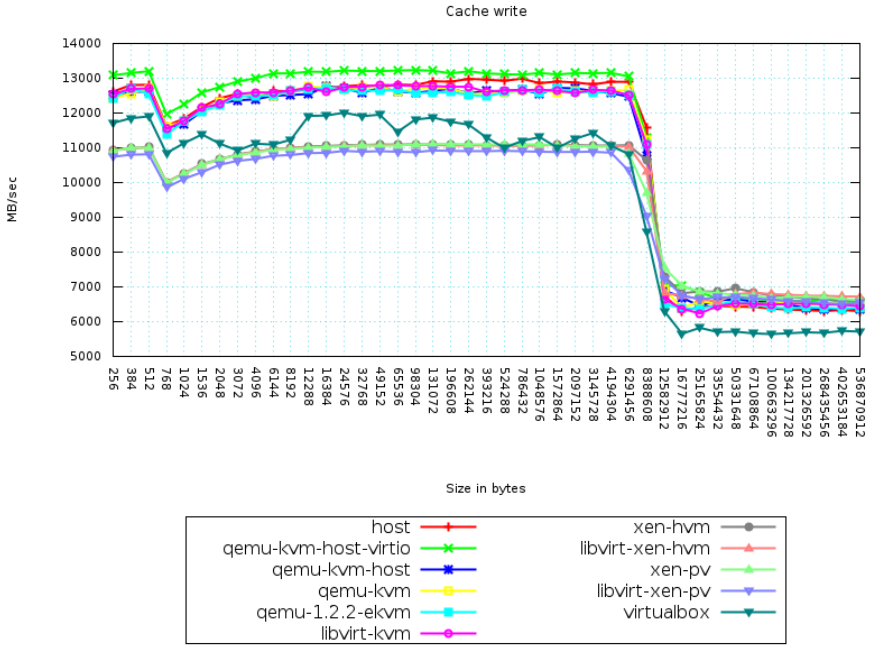

Figure 5.15: Write with 2 cpu cores.

these results that are noteworthy, is the pattern with regard to the various suites that we saw in the read benchmark, which repeats itself here as well. The other, and most interesting, part of these results is how the various suites behave as the size increases. All suites increase slightly for the first three sizes, before they drop about 1000 MB/sec at 768 bytes. They then increase to the same value that they had before the first drop and flatten out. The host, KVM based and virtualbox at about, or just below 13000 MB/sec, and Xen based at about 11000 MB/sec. At the size 8 MB (8388608 bytes) they all drop drastically, and all hover between 6000 and 7000 MB/sec.

The cause of this first drop is somewhat unclear, it happens at 768 bytes, while the L1 cache size is 32 KB, so it is unlikely that it is related to this cache. Meanwhile, the second drop which occurs at 8 MB, which is the same as the L3 cache size, does let itself be explained. This drop is caused by the sample size increasing above the L3 cache size.

Write with 2 cores For the Cache write benchmark with 2 cores enabled in the guests we can see that the pattern we saw with 1 core is quite similar, figure 5.15. We can note a small increase in performance for the KVM-based suites, especially Qemu-kvm-host-virtio that has increased the performance with 500 MB/sec. The four Xen based suites still delivers performance in the region of 11000 MB/sec. VirtualBox on the other hand delivers performance that is lower than it had with 1 core, performance is also in a more erratic manner. Indicating either interrupting processes or an issue with the way VirtualBox handles the operations with 2 cores present.

Comments As can be seen in the above figures the same pattern as with the HPL-benchmark repeats itself, with KVM-based slightly outperforming the Xen-based virtualization suites as well as virtualbox. Another interesting point that emerges from these results is how the KVM-based suites follow the host in performance. Highlighting how close KVM based virtualization is to bare metal performance on the host.

Both the read and write benchmark show no significant difference between the two configurations with 1 and 2 cores enabled in the guests. The greatest difference being produced by Virtualbox, which does behave in a somewhat erratic manner. This might be caused by interrupts from other processes on the host machine, or it can be caused by the way Virtualbox handles critical regions and transitions.

Another interesting result which emerges from both the read and write benchmark, for the most part the write benchmark, is the performance drop which occurs at the size of the L3 cache. All the virtualization suites, as well as the host followed the pattern with this drop, which was quite substantial for the write benchmark. In some cases lowering performance from 13000 MB/sec to about 6500 MB/sec.

5.3.2 LMBench

This section will cover the memory benchmark from the LMBench benchmark suite, firstly I will cover the read benchmark with 1 and 2 cores, before covering the write benchmark with 1 and 2 cores. As has been the case with both the

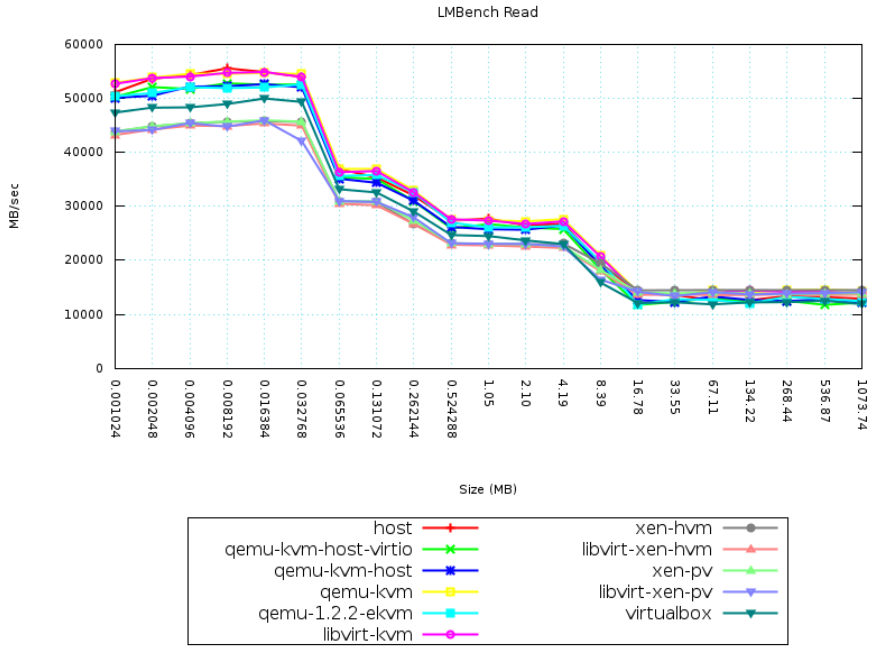

Figure 5.16: LMBench read with 1 core.

HPL benchmark and Cachebench, the results follow the previously established pattern and will be further commented upon.

Also as with Cachebench the figures are presented with the time represented by MB/sec on the Y-axis, and with a size in MB, ranging from 1 kilobyte to 1024 megabytes, on the X-axis. Due to the scale of the graphs the readability is not perfect, for those interested the raw data used for these graphs are included in the appendix.

Read with 1 core The LMBench read benchmark with 1 core in figure 5.16, has the same properties as Cachebench and HPL had, with regard to the varying performance between the host, KVM based, Virtualbox and Xen based virtualization suites. As can be seen in the results, the host and KVM based suites are quite close in their performance. The best performing KVM based suites are, up until the 32 KB size (0.032768 MB), libvirt-kvm and qemu-kvm at about 52000 MB/sec, closely followed by the rest of the KVM based suites with their varying configurations at about 50000 MB/sec. Followed by Virtualbox which lies about 3000 MB/sec below the last mentioned KVM based suites. Lastly we have all the Xen based suites at about 44000 MB/sec.

At about the size 32 KB, we see the first drop in performance, all of the virtualization suites have a performance drop of about 15000 MB/sec. This first performance drop happens when the size is the same as the L1 cache of the host processor. The next drop in performance happens in the range of 128 KB (0.131072) and 512 KB (0.524288), before flattening out. The third and final drop in performance happens when the size is in the range 4 MB (4.19) and 16 MB (16.78), before yet again flattening out. These two last performance drops

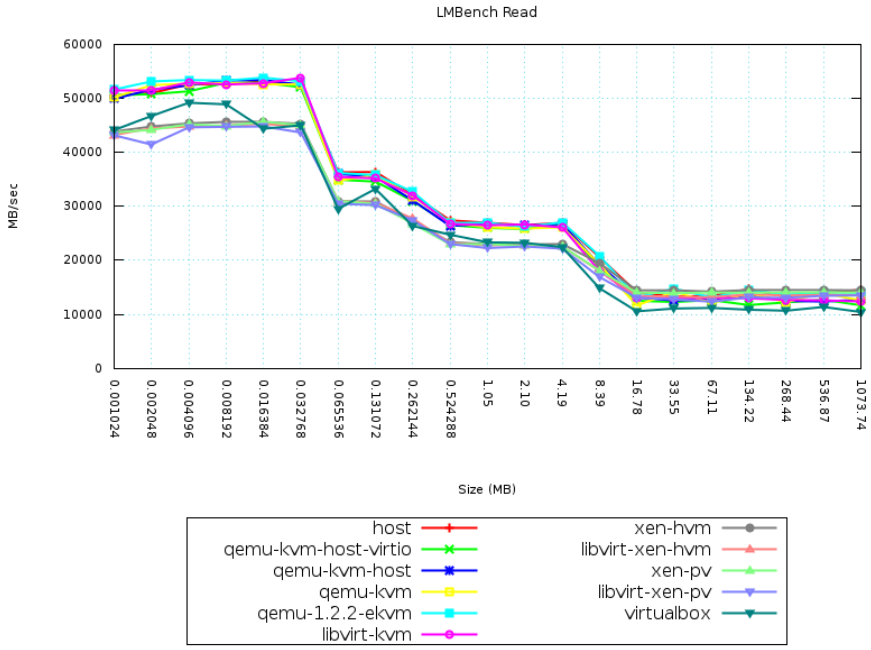

Figure 5.17: LMBench read with 2 cores.

is easily explained, as they both occur at the size of the L2 and L3 cache, whose sizes are 256 KB and 8 MB respectively.

Read with 2 cores The read benchmark with 2 cores repeats the results we saw with 1 core enabled, see figure 5.17. The performance drops does occur at the same places and the general performance of the virtualization suites is the same.

The KVM based suites are still outperforming the Xen based suites, however this time, there is no clear best performer between the KVM based suites. Virtualbox is still situated below KVM in performance, and above the Xen based. The Xen based suites all share the same performance as they did with 1 core, and no suite stand out as being best in performance.

Write with 1 core The LMBench write benchmark with 1 core enabled, share the same properties as the read benchmark did, see figure 5.18. The two KVM based suites libvirt-kvm and qemu-kvm slightly stand out at being the best performers, closely followed by the rest of the KVM based suites. In the other end of the scale we find the Xen based suites, all behaving quite identically with the same performance through the various sizes. Between the KVM based and Xen based we find Virtualbox once again.

As was the case with the read benchmark, the performance drops we saw there are present for the write benchmark as well. First occurring at the size of the L1 cache, 32 KB, dropping to about 30000 MB/sec for the suites. The second drop happens when we get closer to the size of the L2 cache, 256 KB, where the performance drops to about 20000 MB/sec. Before we finally reach the size of

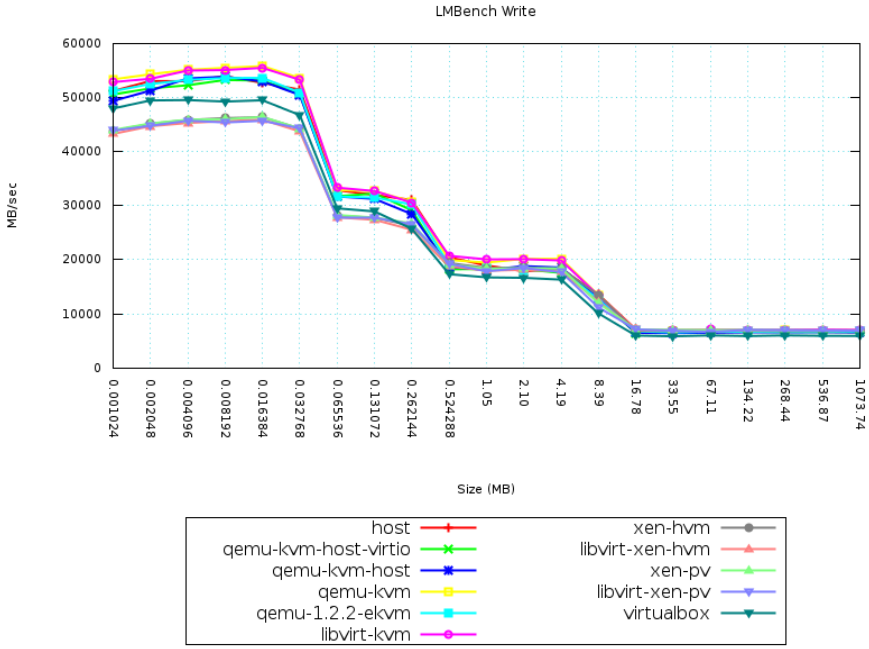

Figure 5.18: LMBench write with 1 core.

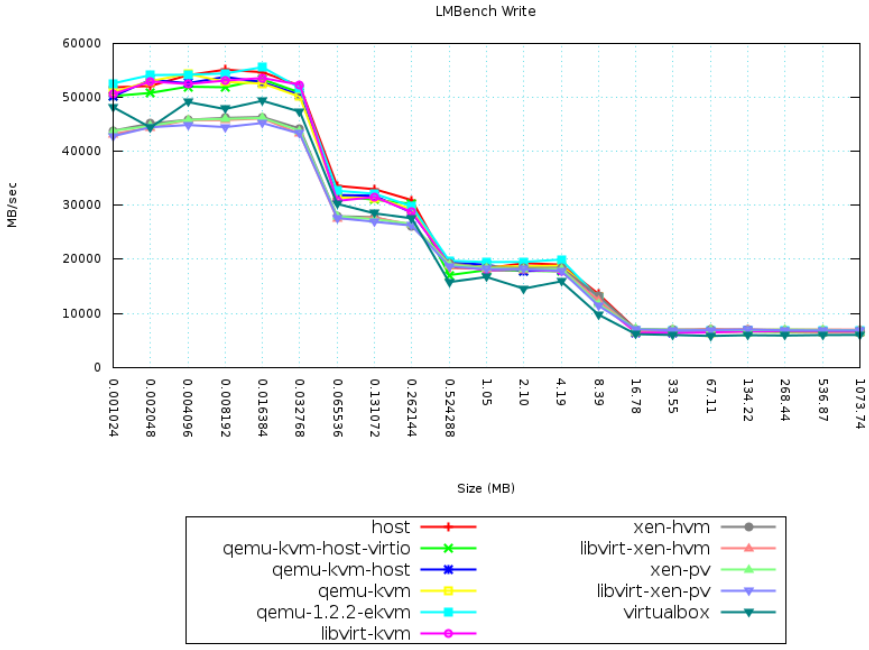

Figure 5.19: LMBench write with 2 cores.

the L3 cache, 8 MB, dropping the performance to about 10000 MB/sec, before the results level out for the remainder of the sizes used in the benchmark.

Write with 2 cores When we look at the results from the LMBench write benchmark with 2 cores enabled, we see little difference from the write benchmark with 1 core, figure 5.19. This time, as was the case with the read benchmarks, the best performing suite is not as clear as it was with 1 core. However, this time the qemu-1.2.2 suite does inch itself slightly above the rest of the KVM based suites, they are all still fairly close the each other the KVM based suites. The Xen based suites does find itself in the other end of the scale yet again, all four still performing quite the same. Virtualbox we find between the two suites, with some jitter in the results here and there.

The performance drops are also still present, and occur at the same sizes as with 1 core. The drops also land at the same performance.

Comments For both the read and write benchmark, with their respective core configuration, we see the same trend in performance across the various virtualization suites. The KVM based performing the best, and even performing the same as the host does, the Xen based suites find themselves in the other end of the scale, while the performance of Virtualbox finds itself consistently between the KVM based and Xen based suites.

Interestingly the performance drops that occur at the various sizes, coincide with the host processors three cache sizes. Performance between the read and write benchmarks have been surprisingly similar throughout the sizes being tested. As the first performance drop at 32 KB occurs, the minor difference between read and write makes itself present. With the difference between read and write being about 5000 MB/sec, from the first drop and throughout the remainder of the benchmark the difference between is about 5000 MB/sec. The last measured sizes for read lies at about 12000 MB/sec, while the write benchmark finds itself at about 6000 MB/sec.

When it comes to individual differences between the various suites, there is a clear pattern, the KVM based performs the best, then Virtualbox, and lastly the Xen based suites. Between the KVM based suites there is some differences in performance, this is most likely caused by the differences in configuration between them, in addition to potential timing issues. When it comes to the Xen based suites, it is interesting to note that they are closer to each other than the KVM based suites are, there are still differences between the Xen based suites just not as prominent as with the KVM based suites. This is for some cases likely caused by the device model which Xen uses, at least for the HVM guests, that is based on a stripped down version of Qemu which handles parts of the memory operations for Xen guests[61].

5.3.3 Comments upon the memory benchmarks

The results produced by both Cachebench and LMBench is interesting and similar in the findings that are presented. They both highlight the performance difference between the KVM based suites, Virtualbox and the Xen based suites. KVM emerging as the best performer for both benchmark suites, while Xen does not perform as well as KVM does, they are still not that far apart.

Both benchmarks measure their domain, Cachebench focuses on the cache hierarchy of the memory subsystem, while LMBench focuses on the memory transfer speeds of read and write operations in this case. Interestingly enough it is LMBench that highlights the speed of the varying caches with regard to the problem size used in the benchmark. Cachebench of course highlighting this as well, with a drastic drop in performance when the size passes the L3 cache size, however this is more prominent in the results from LMBench.

The difference between the suites is again caused by the way the suites handle memory operations. KVM based suites being based on three different versions of Qemu, qemu-1.2.2, qemu-kvm and libvirt using another version of qemu-kvm. Interestingly the Xen based suites which utilizes HVM does use a modified and stripped down version of Qemu to handle memory as well as emulated devices. It is then interesting to note that these two suites based on Xen HVM are not closer to Qemu, while being almost identical in performance to the Xen suites which utilizes paravirtualization.

5.4 I/O-based benchmarks - IOZone

This section will present the results from the IOZone file system and I/O benchmark. The results will be presented as read and write, that has been run on RAW, Qcow and LVM disk configurations. Each of these figures contains six graphs, which each one presents the results with 1 and 2 cores, and the respective file sizes that has been tested. Lastly, for both read and write, I include a summary with all disk configurations and with 1 and 2 cores, using 128 MB file for easier comparison. Please note that the host results are only included for comparison to bare metal performance. The host uses a SSD disk, with LVM partitioning, as well as the EXT4 file system. The guests all use LVM partitioning and the EXT4 file system, which both are the default for a Fedora 17 install.

On the results for Qcow disk image, the reader should note the absence of xen-hvm and xen-pv, which are not able to use a Qcow image for storage. Virtualbox has only used its VDI disk image format, however it is compared to the RAW disk images to get some impression of the performance of Virtualbox.

All results are presented with the record length used on the X-axis, and the performance speed on the Y-axis, given in KB/sec.

Iozone Read Following is the results from the IOzone read benchmark. Firstly we see the results using a RAW disk image, with all configurations present in the figure. Then I will present Qcow and LVM configurations.

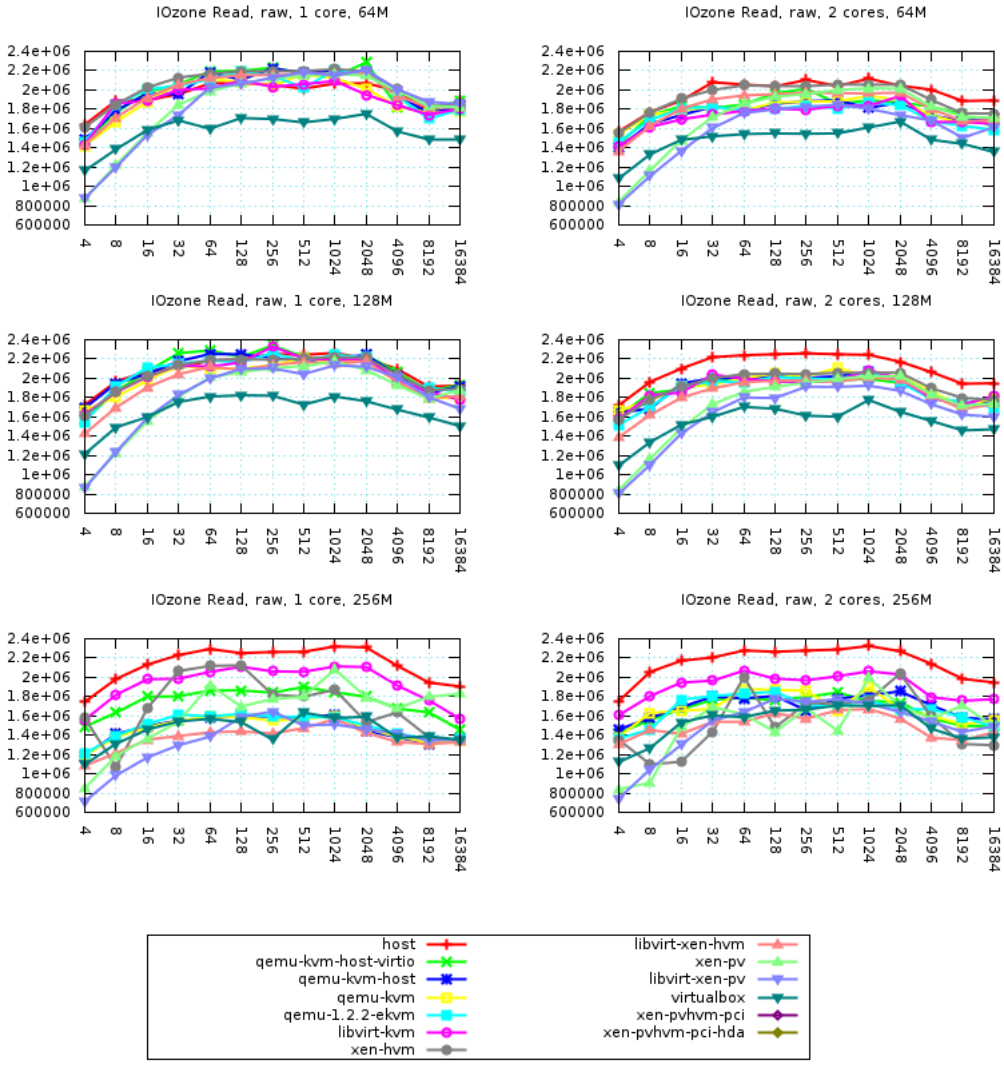

Figure 5.20: IOzone read on RAW disk image.

The results in Figure 5.20 for the RAW disk image does not highlight any clear performance gain for either virtualization suite with 64 MB and 128 MB. When the size is increased to 256 MB we see that the small gap between the suites gets a little bigger. Of the KVM based suites it is, from the plot with 256 MB file size, evident that libvirt-kvm and qemu-kvm-host-virtio are the best performers. Of the Xen based suites, xen-hvm performs better than its three Xen counterparts, however it is not as consistent in its performance. The paravirtualized Xen suites starts their performance with small record lengths quite lower than xen-hvm and the KVM suites. Virtualbox is on average the lowest in performance. Interestingly the performance seems to drop some when increasing from 1 core to 2 cores on the tested systems.

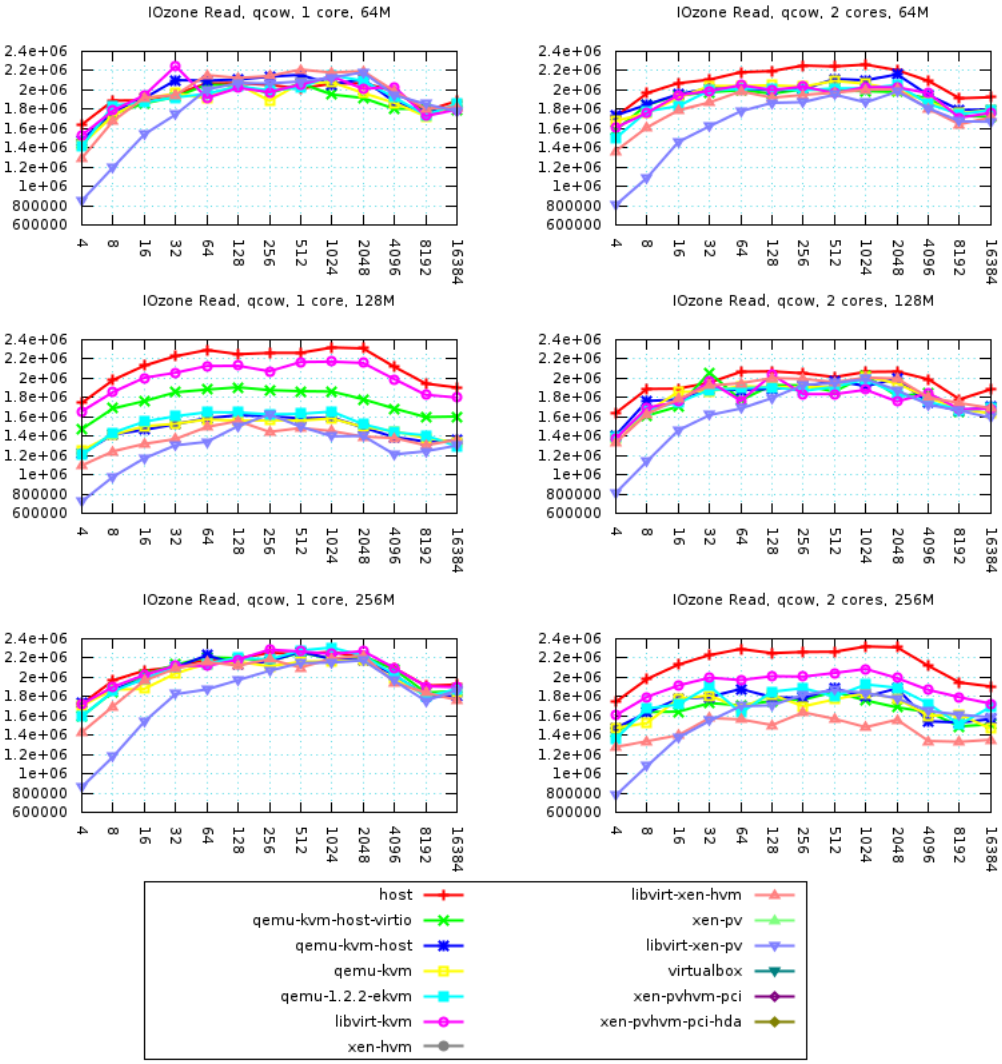

Figure 5.21: IOzone read on Qcow disk image.

The Qcow disk results in Figure 5.21 show more consistency then the results from the RAW disk image, they all behave quite similar and have the same performance development with regard to the record length. Both libvirt-kvm and qemu-kvm-host-virtio do, in what can be best seen in the graphs with 128 MB file and 1 core plus 256 MB file and 2 cores, perform somewhat better than the other suites. The bulk of the suites are situated at the performance range 1.4 GB/sec to 1.8 GB/sec.

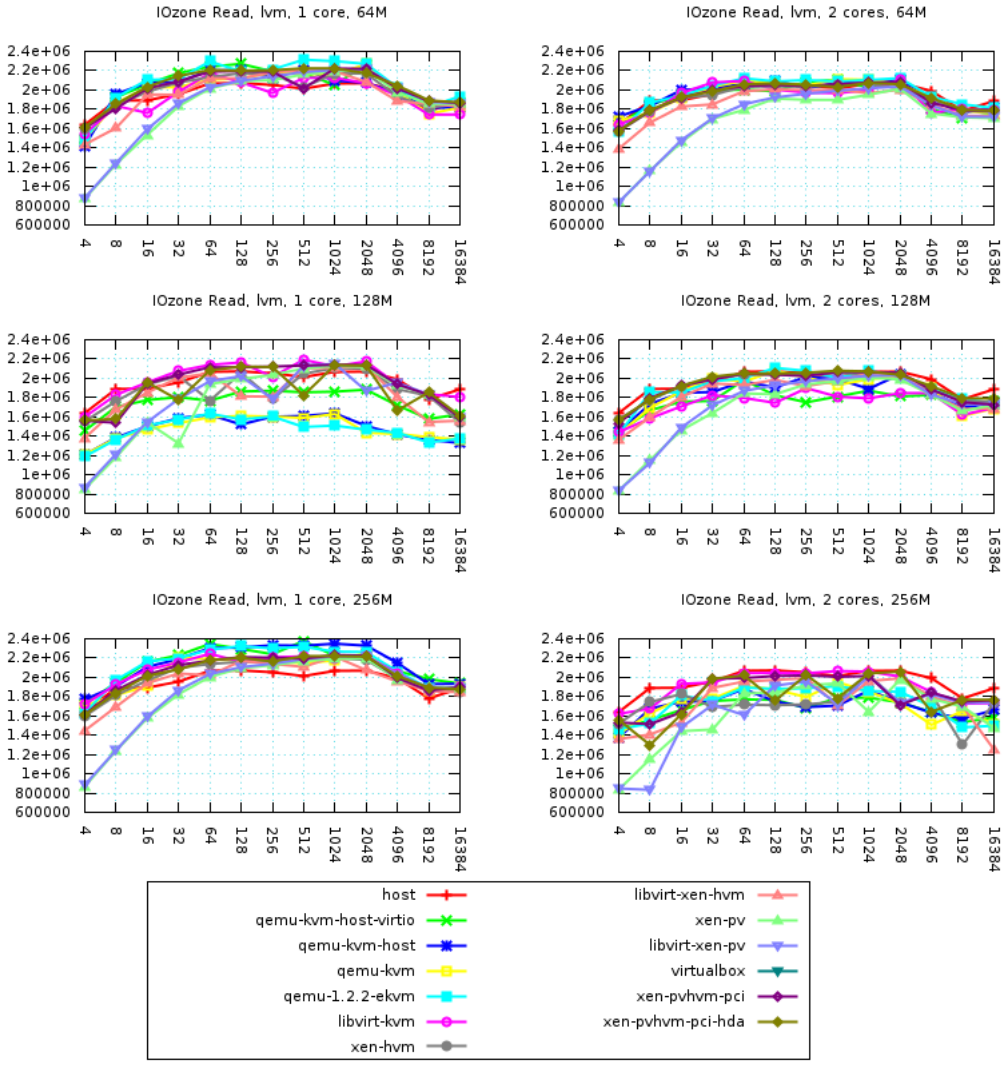

Figure 5.22: IOzone read on LVM disk.

With an LVM disk used for VM storage, as seen in Figure 5.22, it is not the greatest difference in performance than what we saw in the two previous figure. It is noteworthy that performance is best for all suites, and across disk configurations, on LVM with 1 core and a 256 MB file. The same trend for the paravirtualized Xen suites, which starts performance low as we saw for the RAW disk, is repeated here. The best performer for these test all seem, once again to be libvirt-kvm when using 1 core

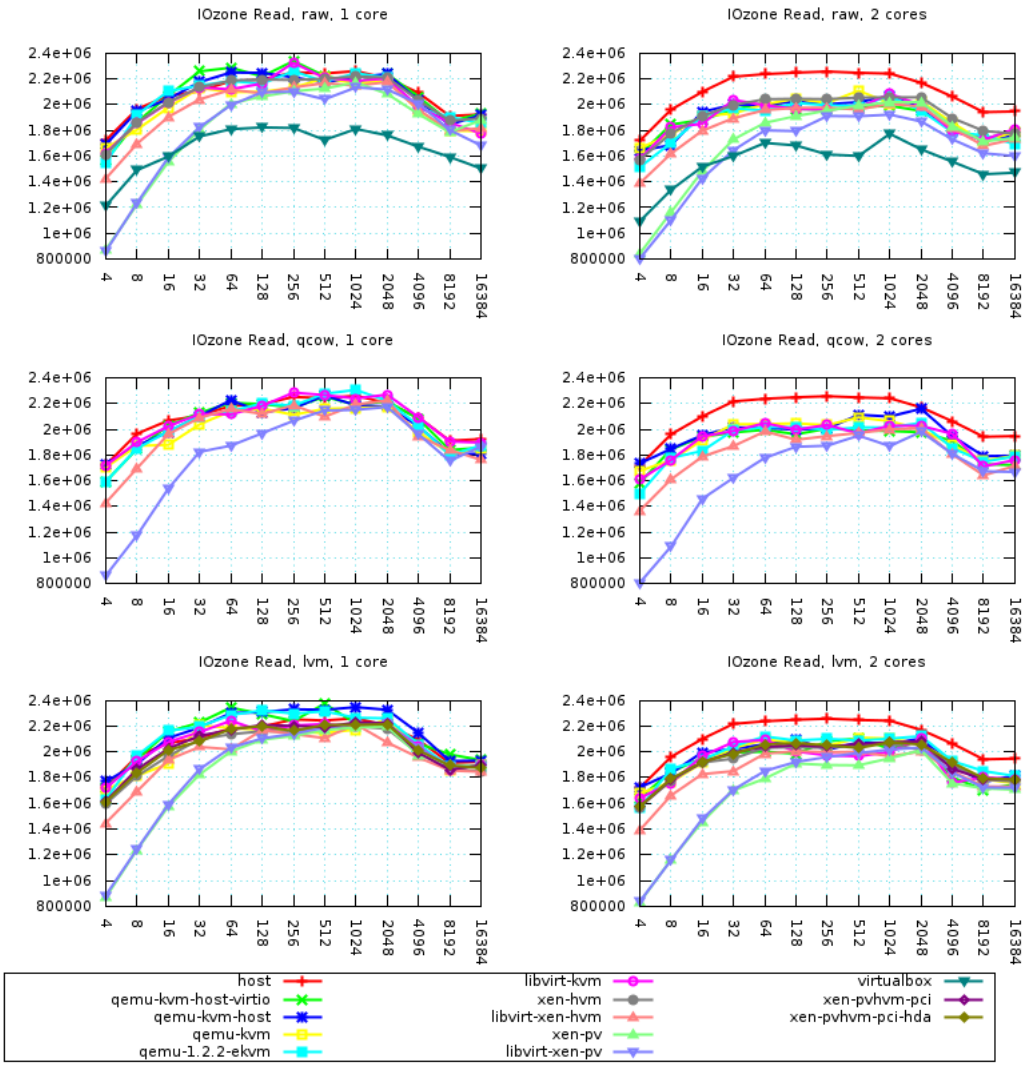

Figure 5.23: IOzone read on all disk configurations with size 128 MB.

Figure 5.23 summarizes the results for IOzone read with file size of 128 for all configurations, and should give some impression as to which disk configuration performs the best for read operations. Both the RAW and Qcow disk configuration behave similarly, with performance being in the same region, peaking at about 2.2 GB/sec with 1 core and about 2 GB/sec with 2 cores. The LVM configuration does have a marginally higher performance with 1 core, but is slower again with 2 cores available.

Of the KVM based suites, the best performer is marginally the libvirt-kvm and qemu-kvm-host-virtio variants. The reason for both of these two to be the best performers is caused by both suites taking usage of Virtio for disk configuration. In the other end of the performance scale, we find Virtualbox, having a 1.8 GB/sec peak performance with 1 and 2 cores. Virtualbox have, as opposed to the other suites, only been tested with a VDI disk image as this is the

default for Virtualbox. Of the Xen based suites the best in performance have been xen-hvm, which has had about the same performance as the KVM based suites. The paravirtualized Xen suites have behaved more strangely, starting with quite low performance with smaller record length sizes, being about 1 GB/sec slower than some of the KVM based suites. Before reaching their peak value at record length 1024 KB, which is just below the KVM based suites.

Iozone Write Following here are the results for the IOZone write benchmark, which will be presented in the same way the the results from read was.

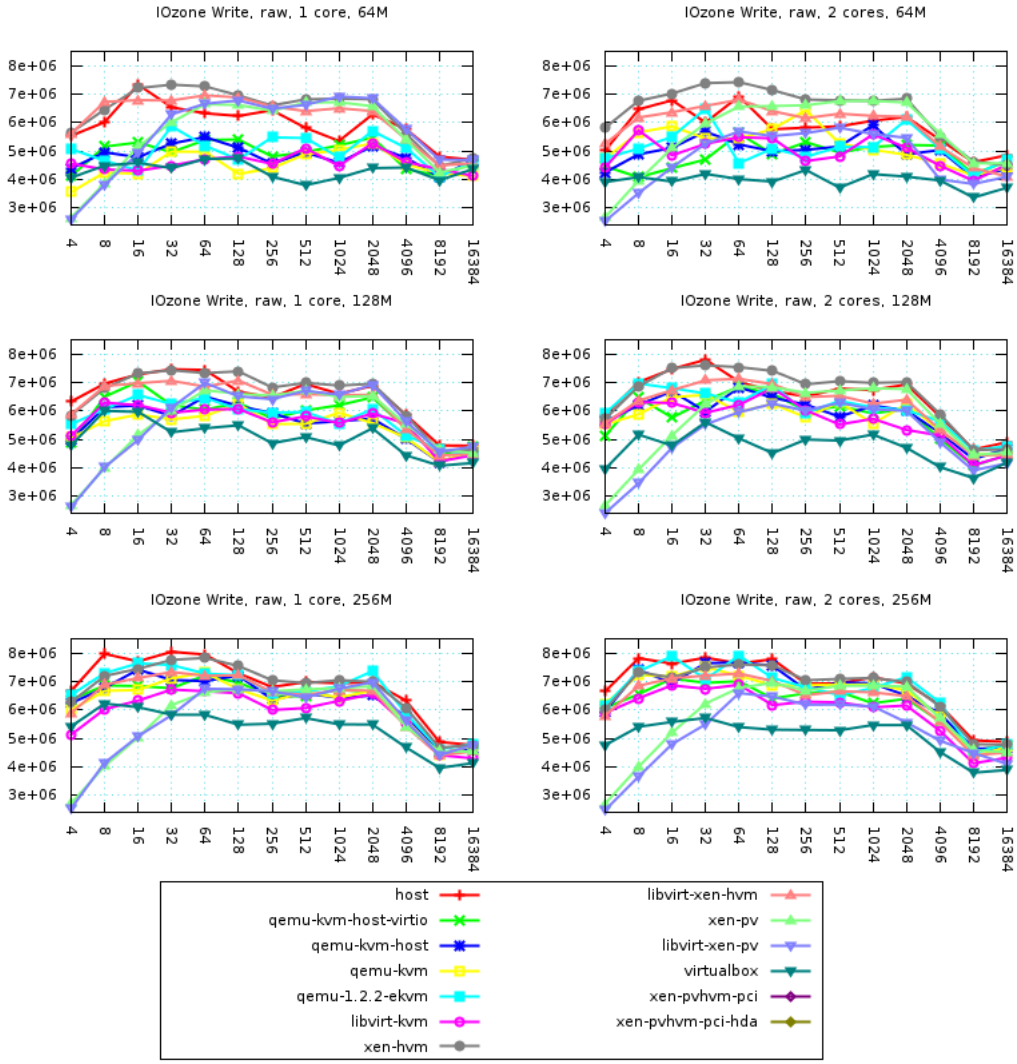

Figure 5.24: IOzone write on RAW disk image.

In the RAW disk image results in Figure 5.24, we see a different trend than what we have witnessed in previous benchmarks, as well as in the read results. In the read results we saw a trend that favored the KVM based suites with the

Virtio drivers. In the write benchmark we see a trend that favors the Xen based suites for all sizes and both 1 and 2 cores. As we did see in the read results, the Xen suites with paravirtualization have a performance that is lower than the other suites tested with regard to the record length. However performance peaks for Xen with paravirtualization with record length being 1024 and 2048. Xen with full virtualization does on the other hand have a higher performance at lower record lengths. Of the KVM based suites qemu-1.2.2 suite performs the best with 1 core for all sizes. With 2 cores it is the qemu-kvm-host-virtio suite that performs the best for all sizes.

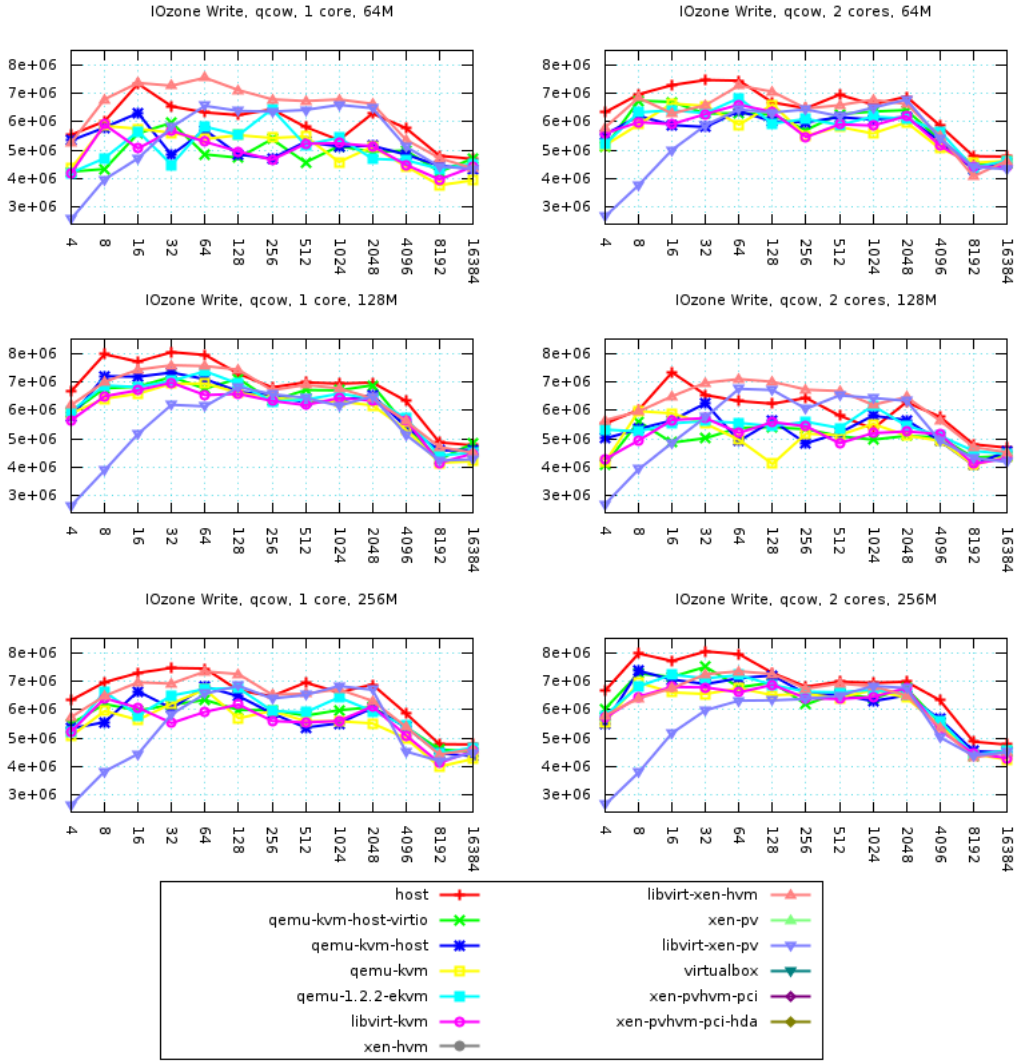

Figure 5.25: IOzone write on Qcow disk image.

When we look at the Qcow disk results in Figure 5.25 we see that Libvirt with Xen fully virtualized guest (libvirt-xen-hvm) is the best performer, with its paravirtualized cousin not behaving quite as good, in addition to the same

low performance for the lower record lengths. The KVM based suites follows the fully virtualized Xen suite for all record length values, albeit not surpassing xen-hvm in performance.

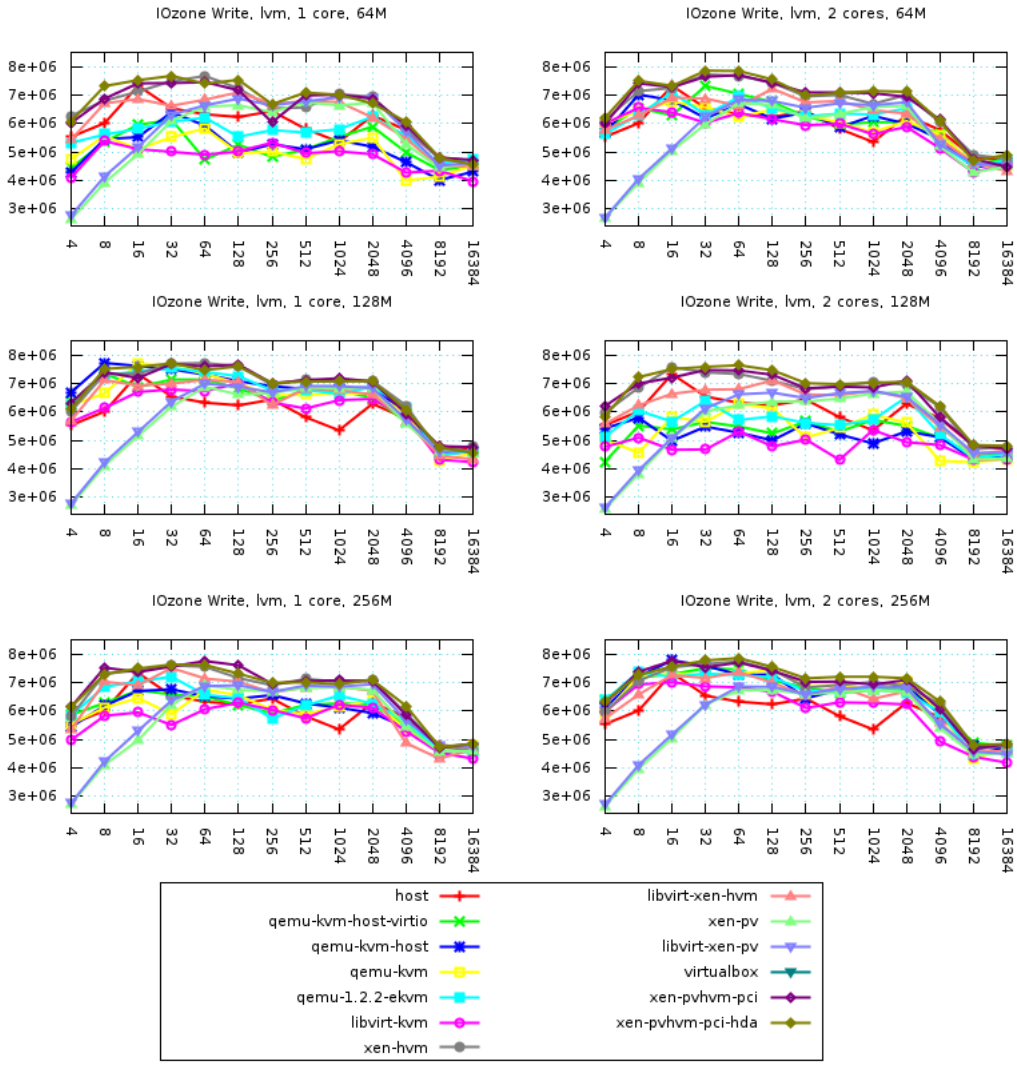

Figure 5.26: IOzone write on LVM disk.

When it comes to the LVM disk configuration that we can see the results from in Figure 5.26. It is quite clear which is the better in performance. Once again a Xen based suite delivers the best performance, this time being represented by Xen with PV on HVM drivers. In more understandable terms, it is fully virtualized Xen guests which utilizes paravirtualization drivers to increase performance, similar to Virtio. This completely bypasses the Qemu device model which Xen uses and provides faster disk performance[62]. All three Xen based suites, xen-hvm, xen-pvhvm-pci and xen-pvhvm-pci-hda delivers decent performance. The paravirtualized Xen suites suffers from the same low performance

for smaller values for record lengths. The KVM based suites performing a bit lower than the Xen HVM and PVHVM suites.

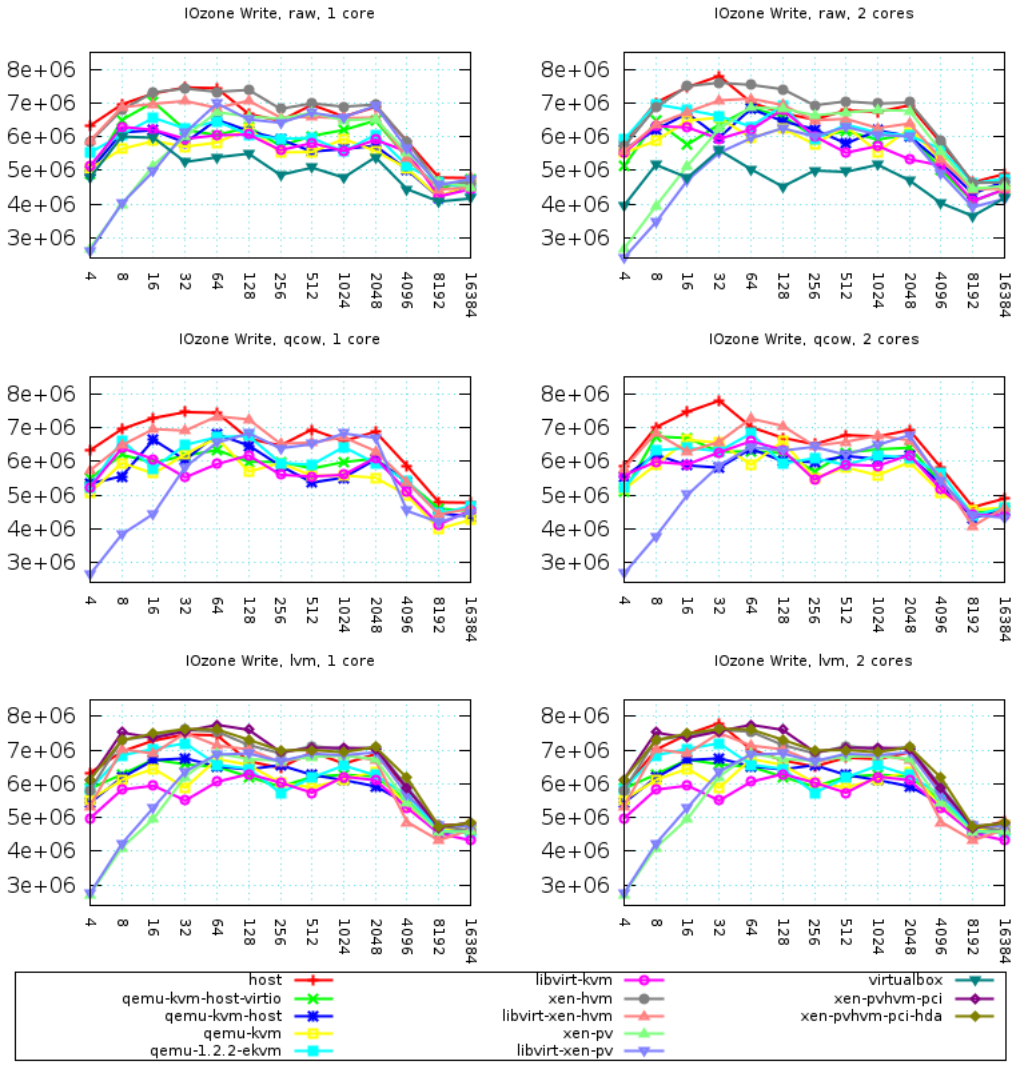

Figure 5.27: IOzone write on all disk configurations with size 128 MB.

Comparing the write results from the various configurations that has been tested yields some interesting observations, as can be seen in Figure 5.27. Based on previously repeated patterns in performance. In these results it is clear that it is the Xen-based suites, especially the fully virtualized that are the best performers. In the RAW disk configuration we see that xen-hvm is clearly outperforming the rest of the virtualization suites that we have seen in these benchmarks, followed by fully virtualized Xen using Libvirt. The paravirtualized suites of Xen have also delivered decent performance, they have however not reached this peak performance before record lengths 1024 and 2048.

When it comes to the Qcow disk configuration it is libvirt-xen-hvm that

performs the best, quite closely to the KVM based suites of which it is qemu-1.2.2 that performs the best among these. libvirt-xen-pv performs on average in this configuration, still struggling with lower record lengths.

Moving to the LVM disk configuration is when things start to get interesting. For these tests I have added fully virtualized Xen with PV on HVM drivers as commented earlier. From the results it is quite clear that disk performance is increased with these options enabled. The drivers actually allow for the guest to have near native disk performance, in some cases it surpasses native performance. Following these added Xen configurations we have fully virtualized Xen, xen-hvm, which performs similarly. When using fully virtualized Xen with Libvirt we actually see a minor drop in performance. The paravirtualized Xen suites still achieves poor performance for the lower values for record lengths, it reaches the fully virtualized suites for values about 1024.

For all three of the disk configurations the KVM based suites have performed decently, yet not on par with Xen utilizing full virtualization. Surprisingly libvirt-kvm, which is one of the two KVM suites that utilize the Virtio drivers, has been located in the lower performing of the KVM based suites for all write configurations.

5.4.1 Comments

As shown in the I/O benchmark results, there are a few differences between the various suites that have been tested. For the read operations there are some differences with regard to the number of processor cores present. With the write benchmarks this difference in performance with the number of cores seem to be non-present.

In the read benchmarks there is no definitive performance winner, overall it would seem that libvirt-kvm and xen-hvm is the suites that does perform the best. However the other suites are closely located to both. Looking at the write benchmark results, it is the xen-hvm suite that stands out, along with the xen-hvm variations with PV on HVM drivers enabled. Performance of these three are near native.

An issue with the paravirtualized Xen suites has also become apparent, both perform well below the other suites when the record length is low. This performance does quickly rise, and with the record length at 512 to 2048 it performs at the same level as Xen with full virtualization.

VirtualBox does suffer in these benchmarks, it performs the lowest of all virtualization suites tested, it is also the only suite that has used the VDI disk image format. It is for that reason worth to note that a Virtualbox configuration set to use either RAW, Qcow or an LVM disk might perform equally to both the Xen based suites and the KVM based.

For both the read and write benchmarks there is a performance drop that occurs when the record length becomes 8192 KB and increases. This does coincide with the L3 cache. The write benchmarks do also have a performance drop that occurs when the record length reach 256 KB, which is the same as the L2 cache. From the read benchmarks we can see in the previous figures that all suites perform similarly after the performance drop at 8192. Looking at the write benchmarks it is the Xen based suites that performs the best after the performance drop at 256, and at 8192 all suites have similar performance.

Interestingly Xen performs the same as KVM for read operations, while on write operations Xen surpasses KVM. The results are most interesting when the LVM configuration is used, this is where the fully virtualized guest also has used the PV on HVM drivers. From these results the three Xen guests, xen-hvm, xen-pvhvm-pci and xen-pvhvm-pci-hda, outperforms the other virtualization suites. One reason for the PV on HVM configured Xen suites to perform so well as they do is likely to be the missing overhead of the Qemu device model. In these two the Qemu device model is completely bypassed and enables faster disk operations. For the paravirtualized Xen guests it is interesting that they do not perform as well as the two PVHVM guests, given that they all take use of the same paravirtualized drivers.

For the KVM based suites it is also interesting to see that the guests that utilize the Virtio driver do not perform better than they do. For the write benchmarks libvirt-kvm performed the lowest of the KVM based suites, and it does utilize the Virtio driver. While qemu-kvm-host-virtio performed on average along with the other KVM suites.

Chapter 6

Conclusion

6.1 About the conclusion

This chapter will conclude this thesis, and also conclude the last two chapters. Firstly I will comment upon what we learned in Chapter 3. Before we move on to the results based on Chapter 4 and presented in Chapter 5. Before this chapter draws to a close I will comment upon the shortcomings of this thesis, and what has gone wrong about the benchmarks performed. Lastly I will comment upon future work that would be interesting to see the results from, however that is outside the time-frame of this thesis.

6.2 Virtualization software

In Chapter 3 we looked at the most prominent virtualization technology in the open-source world. We saw how they compared architecturally and in terms of usage. Architecturally Xen is the most intrusive of the virtualization suites replacing the core parts of the host operating system and placing itself on top of the hardware. While KVM and VirtualBox retains an architecture that uses a kernel modules to interact with the hardware, making it less intrusive. Xen might be, for its hypervisor architecture, better suited in a server environment and used as a virtualization server. KVM and VirtualBox on the other hand are both suited for use in a server environment and as virtualization on the desktop.

The usage of the virtualization suites we have seen relies on the skills and requirements of the user. All can be configured from the command line making them suitable for use in servers and machines that do not necessarily have a display connected. For system administrators with little confidence with a command line, the use of Libvirt and VirtualBox is probably the best choice. With a graphical front-end known as Virtual Machine Manager, Libvirt can utilize both Xen and KVM virtualization with a graphical interface. And VirtualBox being by far the most graphical oriented suite of all the suites we have seen.

6.3 Benchmarking results

6.3.1 CPU-based benchmarks

As we saw in the previous chapter, there is a clear indication that KVM based virtualization suites has surpassed Xen in terms of performance. First we saw that the High Performance Linpack benchmark resulted in Qemu-KVM to perform the best, with a performance that could compete with bare metal performance. However it would be interesting to see whether this still would be the case with more than one virtualization suite present. With regard to the findings in[10] it is interesting to see that performance of KVM has surpassed Xen, and even performs quite closely to the host performance. Of the Xen based suites, we see that both that are tested with the `xl` tools and not Libvirt performs the best (xen-hvm and xen-pv), interestingly these are full virtualization and paravirtualization respectively. At the end of the performance scale we find Virtualbox. Which does deliver performance that is a stretch away from the KVM based suites, but that is similar in performance to the Xen based suites which utilize Libvirt.

When we look at the context switch results the results are divided. With 1 and 2 cores for the guests, it is clear that KVM performs the best. When the number of cores is increased we see that Xen performs better and more consistently. Highlighting the architectural differences between the two virtualization suites in terms of scheduling. Interestingly, fully virtualized Xen using the `xl` tool-stack delivers performance that is close to the KVM-based suites for 1 and 2 cores, and performs the best for 4 and 8 cores.

Looking at Virtualbox at size 0 for all context switch benchmarks, we can see that the results are different for various configurations. With LMBench using 1 core Virtualbox delivers performance between the three aforementioned Xen suites and the KVM based suites. When the number of processor cores in the guest is increased we see that the time used for a context switch is drastically increased. If we are to look at the results from Context Switch we see that this drastic increase in context switch time has diminished.

The most interesting numbers that we have seen from these results, with regard to both context switch benchmarks, is the time taken by the host to make a context switch. What is assumed is that the host should deliver the lowest numbers in both benchmarks and be the fastest at context switching. The figures presented suggest otherwise.

Benchmarks done in [10] and [11] with context switching suggested that the host would deliver performance that is faster than the VMs running on the same hardware. The cause of this difference is in all likelihood the presence of hardware assisted paging on the host processor. Also known as Intel EPT and AMD RVI. As pointed out earlier, the presence of these additions can increase performance by 48% with a MMU-intensive benchmark and 600% with a MMU-intensive microbenchmark in the case of Intel EPT.

6.3.2 Memory-based benchmarks

The memory based benchmarks measured performance in two ways, Cachebench focused on the cache memory hierarchy, while LMBench focused on the memory bandwidth for a number of operations. Both were tested with read and write

operations.

Cachebench presents results that for both the read and write benchmarks suggest that KVM achieves higher performance with regard to cached memory operations. For the read benchmark, KVM delivers results in the region of 2.5-2.6/Gbps, while Xen delivers read speeds at about 2.2/Gbps. Between the two we find Virtualbox where the performance delivered is in the region of 2.3-2.4/Gbps. All results have an indication of a slight performance drop when the size of the level 3 (L3) cache is reached in sampled size.

The write benchmark delivers the same indication as the read benchmark did. KVM based suites delivering near native performance, while the Xen based suites have a slightly lower performance. Where the KVM based suites, as well as Virtualbox, delivers a performance in the region of 12-13/Gbps. Xen is in the region of 10-11/Gbps. All the tested virtualization suites experience a performance drop for write operations when the size of the L3 cache is reached as well.

The results from LMBench read and write benchmarks highlight the same findings as we saw in the Cachebench results. The KVM based suites slightly outperform the Xen based suites and Virtualbox. Performance degradation with regard to tested size also giving an indication of the speed of the three levels of cache present on the host system.

All results from both Cachebench and LMBench highlights the performance of KVM having surpassed Xen in memory and cache operations, also delivering near native performance. Both KVM and Xen utilize the hardware features for MMU management or hardware assisted paging (Intels EPT and AMDs RVI). Allowing for greater performance when shadow page-tables can be handled in hardware and not in software. Results in [10] and [11] did suggest equal performance between KVM and Xen, however the then version of KVM lacked EPT support on Intel platforms. While over the years since then this feature has not only been added to KVM, it has been optimized as well. As a result KVM has now surpassed Xen in memory operations and delivers performance that is more or less equal to the performance of the host.

6.3.3 I/O-based benchmarks

Where the two previous sets of benchmarks suggested that the KVM based suites delivered the best performance, then looking at the results from IOZone suggested otherwise. From an overall perspective it was Xen with full virtualization that emerged as the best performer, and Xen with paravirtualization delivering equal performance when the record length tested was increased.

For the results from the read operations that were delivered from the tested virtualization suites it was apparent that the performance was quite equal across the tested suites, except for Virtualbox. Both the RAW disk image and the LVM disk configuration favored the Xen full virtualization suites. While the Qemu developed Qcow disk image format favored, unsurprisingly, the KVM based suites. Of the KVM suites it was the two that utilize Virtio that emerged as the best performers.

While for the write operation results, it was clear that the favored virtualization suite on all tested disk configuration was Xen based, both with and without optimization to the disk parameters. The gained performance of Xen can also be attributed to the stripped down version of Qemu which handles I/O

in the fully virtualized Xen suite, and also the complete lack of Qemu in the two optimized configurations and the paravirtualized Xen suites. The two KVM suites with Virtio did not perform as well for the write operations as they did for the read operations.

As we saw in [67], where the results suggested that KVM and Xen delivered quite equal performance, while delivering performance below the physical host. This is not necessarily the case for the results presented in this thesis. All suites have delivered performance that is more or less equal to the host, with Xen being the best performer.

Since the aforementioned paper was published, we have seen the emergence of Virtio for Linux, mainly targeted at KVM, that would also increase the disk performance. We have also seen the emergence of I/O MMU virtualization technology, such as Intel VT-d and AMD-Vi, which allows guests to directly use peripheral devices such as hard disks. Thus making it possible to bypass the software translation for an I/O operation, enabling the guest to directly access hardware resources[1]. As well we have seen the popularization of Solid State Drive (SSD) which have been used on the host computer for the benchmarks.

The two KVM suites that utilize Virtio have delivered decent performance in the read benchmarks, while not being at the top for the write benchmarks. While Xen using full virtualization and with optimizations emerge as the best in performance.

6.3.4 Final words

With the rapid development of both KVM and Qemu, and a growing community, it is not surprising that KVM in many cases as surpassed Xen in terms of performance. As we have seen KVM can achieve up to near native CPU performance, as well delivering performance of memory operations that is near equal to the host performance. While the I/O based benchmarks suggested that the difference is not all that great, and in some cases Xen performs better than what KVM does, especially for write heavy file operations.

There is no definitive answer as to which virtualization suite is best if looking only at performance. The KVM based suites favor processor and memory heavy workloads, and are probably best suited for workloads and users where raw processor power is of importance. While Xen seems to favor the I/O heavy workloads, especially those that have high importance of write operations. The results does indicate that optimizations and drivers such as Virtio for KVM and the PV-on-HVM drivers for Xen, benefits the performance.

6.4 Shortcomings

Sample size

The sample size that has been used is five, from which the benchmarks have been run for five iterations and producing a mean. A bigger sample size should have been used, at least ten. However the number of platforms that has been tested and the amount of time that some of the benchmarks has used. The time

[1] IOMMU has however not been used for the benchmarks in this thesis.

used to benchmark one virtualization platform would simply be enormous and not possible within my time-frame.

Context Switches

While the results that were gathered from the context switch benchmarks did present some interesting findings. The configuration for the Context Switching benchmark should have been redone. This because of an error that assumed that the input for the arrays size was given in kilobytes and not bytes as was the case. This means that the benchmark has been run with an array size that is far to small to fill the systems caches and measure the cost of a context switch when the entire cache has to be flushed.

For the LMBench context switch results the same applies to some extent. While for two processes the cache is never entirely filled, when more processes are used the cache gets filled up with more data. Giving some indication as to the cost of a context switch when the cache does get filled.

IOZone

The only remark of the IOZone benchmark is that it should have been run with more sizes for the sample file, i.e. from 64 MB to 2048 MB (2 GB, the VMs memory size).

VirtualBox

In all of the virtualization platforms that was benchmarked, the configuration of the VMs where configured as close to default as possible. For this to be the case for VirtualBox it meant to use the VDI disk image format which is default for VirtualBox. While VirtualBox does support raw block devices and disk images such as an LVM partition and a RAW disk image, in addition to the Qcow2 disk image configuration. It is not assumed that most users of VirtualBox would use those options, raw block devices for instance requires some configuration to be enabled for guests. This is a shortcoming for the benchmarks on VirtualBox.

6.5 Future work

For the benchmarks and measurements that have been performed in this thesis there is some benchmarks and considerations that could have been used, however that were not possible within the time-frame and due to the amount of that would have been generated.

The equipment used for the benchmarks have been a desktop computer, for future benchmarks it would be interesting to see results from server hardware.

Virtualization software also allows the user to pin a processor core to a virtual CPU in a guest. This is said to increase the performance of guests, since the data in the cache is the same for the VCPU and host CPU. It would be interesting to see the possible performance gain. And if the virtualization suites in this thesis would benefit from pinning the VCPUs and possible how the increased performance compares.

Another important factor that was not measured in this thesis is scalability. This is measured by Xu et al. in[67], of which Xen did scale the best. With

the development since then, it would be very interesting to see if this still is the case.

Appendix A

Additional results

A.1 About

This chapter will present some of the data that have been previously presented in table form, this is due to readability issues with some of the graphs.

A.2 LMBench CTX

Size	Procs	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0	2	1.196	1.092	3.704	1.14	3.628	1.262	3.682	5.846	1.08	1.118	1.508
0	4	1.166	1.112	3.73	1.156	3.77	1.326	3.76	6.512	1.16	1.198	1.66
0	8	1.252	1.134	3.848	1.21	3.88	1.334	3.892	10.704	1.22	1.196	1.744
0	16	1.3	1.218	4.008	1.272	4.148	1.412	4.112	6.698	1.264	1.242	1.896
8	2	1.212	1.182	3.896	1.262	3.898	1.364	3.808	6.702	1.196	1.212	1.672
8	4	1.392	1.346	4.012	1.364	3.974	1.57	4.012	13.296	1.384	1.404	1.894
8	8	1.442	1.338	4.22	1.406	4.094	1.548	4.254	8.222	1.422	1.388	2.138
8	16	1.656	1.562	4.508	1.66	4.438	1.78	4.522	8.02	1.634	1.604	2.598
16	2	1.32	1.248	4.014	1.324	4.076	1.454	3.986	5.196	1.294	1.262	1.796
16	4	1.43	1.46	4.252	1.434	4.238	1.69	4.166	6.796	1.414	1.516	2.114
16	8	1.558	1.526	4.524	1.686	4.632	1.73	4.63	8.136	1.658	1.554	2.56
16	16	1.874	1.77	4.774	1.918	4.9	2.034	4.794	10.182	1.864	1.81	3.07
32	2	1.31	1.208	3.516	1.334	3.776	1.41	4.012	7.698	1.348	1.298	2.052
32	4	1.556	1.598	4.652	1.594	4.644	1.872	4.55	10.414	1.612	1.668	2.508
32	8	1.954	1.886	4.892	2.018	4.74	2.11	4.986	21.722	1.956	1.878	3.222
32	16	2.086	1.95	4.982	2.048	4.97	2.248	5.06	26.046	2.086	2.04	3.416
64	2	1.63	1.478	4.134	1.716	4.776	1.526	4.414	10.202	1.598	1.534	2.81
64	4	2.236	2.266	5.778	2.414	5.716	2.524	5.454	19.734	2.25	2.198	3.87
64	8	2.518	2.354	5.502	2.574	5.448	2.654	5.604	12.394	2.66	2.56	4.344
64	16	2.498	2.37	5.524	2.554	5.608	2.7	5.646	23.596	2.474	2.362	4.396
128	2	2.73	2.52	5.418	2.794	5.304	2.852	6.174	19.508	2.874	2.416	4.952
128	4	3.124	3.076	7.232	3.292	7.852	3.604	7.024	44.128	3.246	3.214	5.368
128	8	3.104	2.94	6.568	3.176	6.572	3.452	6.84	52.37	3.218	2.942	5.448
128	16	3.014	2.918	6.428	3.076	6.414	3.328	6.528	49.078	2.94	2.886	5.432
256	2	3.344	2.652	6.008	3.036	6.446	3.144	7.012	69.05	2.846	2.788	6.64
256	4	3.756	3.198	7.522	3.136	7.726	3.596	7.928	74.11	3.184	3.336	6.468
256	8	3.262	2.816	6.79	3.374	7.476	3.36	7.836	74.268	3.2	2.986	6.6
256	16	3.338	3.076	7.004	3.058	6.86	3.676	7.912	68.37	3.302	3.496	6.916
512	2	3.014	2.458	5.482	2.522	6.204	2.956	6.57	108.668	2.73	3.142	7.092
512	4	3.262	2.666	7.034	2.676	5.228	3.522	7.59	108.596	3.124	3.342	7.906
512	8	3.288	2.674	6.564	3.29	6.624	3.702	7.938	111.738	2.57	2.808	8.526
512	16	14.356	13.654	16.468	14.904	19.924	12.938	18.004	113.542	15.292	14.492	21.166
1024	2	2.368	2.586	5.54	3.042	5.5825	3.706	6.846	179.576	3.04	3.192	14.756
1024	4	4.35	3.116	7.2	2.932	7.274	4.648	8.108	190.194	3.08	4.964	12.39
1024	8	24.616	22.45	24.872	22.166	28.296	21.056	26.266	199.952	24.726	23.208	33.628
1024	16	41.934	39.48	39.012	42.048	38.076	35.768	40.096	214.02	43.198	41.646	52.84

Figure A.1: LMBench CTX with 1 core.

Size	Procs	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0	2	1.224	1.148	8.196	1.204	6.694	1.416	4.476	6.5	1.102	1.218	44.89
0	4	1.228	1.196	8.482	1.242	8.984	1.43	4.056	6.336	1.228	1.334	46.298
0	8	1.322	1.268	9.114	1.16	9.138	1.45	4.814	9.866	1.252	1.352	45.778
0	16	1.394	1.358	9.8	1.244	8.96	1.496	4.494	8.998	1.322	1.41	44.814
8	2	1.294	1.172	7.292	1.236	7.802	1.448	4.766	5.952	1.248	1.266	45.986
8	4	1.474	1.312	8.752	1.322	8.926	1.596	4.81	13.878	1.292	1.422	46.69
8	8	1.516	1.49	8.986	1.43	9.594	1.614	4.75	8.188	1.434	1.526	47.312
8	16	1.784	1.684	9.168	1.614	9.94	1.87	5.074	10.866	1.596	1.788	47.532
16	2	1.352	1.3	6.93	1.234	8.284	1.624	5.15	5.39	1.238	1.302	45.052
16	4	1.56	1.428	9.73	1.424	10.93	1.668	5.078	11.282	1.364	1.534	49.02
16	8	1.75	1.786	10.236	1.63	10.03	1.82	4.854	14.62	1.724	1.784	45.996
16	16	2.05	2.05	10.236	1.878	10.942	2.138	5.482	9.58	1.906	2.012	48.162
32	2	1.488	1.282	8.304	1.258	7.848	1.5	5.162	8.29	1.396	1.536	46.194
32	4	1.71	1.63	11.348	1.616	11.494	1.902	5.984	23.602	1.652	1.722	48.33
32	8	2.084	2.126	11.0	2.074	10.95	2.288	5.168	9.47	2.064	2.092	49.21
32	16	2.258	2.462	10.67	2.346	11.598	2.428	5.652	20.42	2.19	2.184	49.178
64	2	1.484	1.594	8.136	1.688	9.726	1.896	5.794	9.348	1.646	1.564	48.524
64	4	2.502	2.3	11.086	2.262	12.452	2.566	6.062	34.288	2.372	2.41	51.678
64	8	2.984	2.786	11.368	2.664	11.362	2.942	5.922	19.988	2.852	2.856	52.19
64	16	2.838	2.732	11.462	2.624	11.486	2.944	5.938	28.26	2.904	2.94	52.422
128	2	3.106	2.798	10.482	2.98	10.414	3.002	7.888	21.36	2.714	2.646	52.886
128	4	3.502	2.81	12.866	3.142	13.308	3.684	7.286	46.22	3.096	3.182	55.304
128	8	3.692	3.258	13.06	3.168	13.456	3.816	7.172	53.066	3.762	3.344	55.452
128	16	3.37	3.22	12.896	3.074	12.656	3.672	7.36	51.616	3.448	3.3	54.136
256	2	2.916	2.95	10.132	3.008	9.756	3.418	8.458	65.394	2.55	2.724	58.256
256	4	3.85	2.902	13.708	3.276	13.568	3.83	8.694	74.128	3.16	3.586	60.24
256	8	3.112	3.604	12.802	3.342	12.508	3.762	8.578	74.166	4.014	3.896	86.202
256	16	3.692	4.844	13.67	3.56	14.852	4.358	8.232	67.702	3.678	4.32	60.518
512	2	2.868	2.025	13.04	2.854	13.432	3.446	8.704	104.58	2.1625	3.3025	143.522
512	4	3.548	3.882	14.794	3.144	14.66	4.002	8.674	111.702	2.7425	4.122	148.764
512	8	3.682	2.83	15.986	3.136	17.288	4.302	8.924	111.16	2.442	4.008	118.53
512	16	15.462	17.172	25.452	13.902	25.19	14.03	18.754	114.13	14.104	15.528	167.3
1024	2	4.54	4.13	20.994	3.222	15.672	4.244	8.04	189.25	4.5575	3.6425	280.634
1024	4	4.44	6.258	28.928	3.964	20.896	5.648	10.334	185.426	3.798	5.634	332.2
1024	8	24.74	25.654	53.46	21.654	44.628	21.71	27.476	200.578	25.294	28.288	379.814
1024	16	46.316	50.184	65.134	39.56	59.044	36.818	42.85	204.856	45.832	43.346	411.05

Figure A.2: LMBench CTX with 2 cores.

Size	Procs	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0	2	11.99	1.17	7.118	12.284	6.946	9.652	6.716	6.5	37.336	32.032	44.758
0	4	18.46	11.188	7.368	3.606	7.758	7.648	6.476	6.336	12.366	1.336	129.19
0	8	4.446	4.748	9.176	4.928	10.026	5.182	6.534	9.866	5.906	1.674	128.586
0	16	7.194	10.43	9.762	4.454	10.486	3.96	6.448	8.998	8.976	2.388	127.332
8	2	32.066	33.83	7.606	22.758	6.768	11.034	8.184	5.952	26.152	4.764	45.29
8	4	1.492	2.42	8.216	1.366	7.862	4.798	7.516	13.878	2.104	4.222	127.324
8	8	6.244	8.326	10.454	5.014	11.086	5.49	8.012	8.188	9.47	1.592	133.496
8	16	3.124	6.314	10.412	5.216	11.112	8.448	6.64	10.866	6.106	9.234	127.488
16	2	33.354	22.702	6.812	34.862	6.866	10.65	6.662	5.39	26.054	1.44	60.256
16	4	24.544	1.464	8.75	1.68	9.034	7.24	6.37	11.282	12.85	6.874	130.958
16	8	3.692	1.75	10.714	7.58	11.052	6.536	8.458	14.62	2.564	3.642	141.278
16	16	14.2	4.126	11.308	4.092	11.092	8.432	7.524	9.58	8.574	5.764	138.908
32	2	28.252	17.28	7.552	49.366	7.564	9.762	9.61	8.29	28.368	1.378	48.352
32	4	1.836	31.426	8.472	1.752	8.962	8.09	7.9	23.602	1.72	13.028	134.966
32	8	3.9	10.244	11.42	8.474	11.154	7.69	8.866	9.47	6.89	2.778	139.908
32	16	10.31	5.998	11.908	9.632	12.006	10.426	7.734	20.42	15.49	4.254	146.066
64	2	56.114	14.032	8.92	53.084	9.054	11.748	12.756	9.348	46.798	27.358	50.046
64	4	11.178	3.284	10.288	2.394	10.012	7.674	6.806	34.288	2.3	2.782	146.786
64	8	14.368	7.816	14.026	4.624	12.996	8.666	9.612	19.988	18.08	3.362	151.822
64	16	15.756	12.718	12.848	9.126	14.3	6.948	9.09	28.26	11.574	4.982	150.34
128	2	44.998	31.622	10.404	18.514	11.17	11.766	12.61	21.36	23.906	28.516	52.346
128	4	22.824	20.33	13.87	14.998	14.874	5.992	9.106	46.22	3.452	19.102	166.79
128	8	18.12	19.88	16.098	30.662	16.38	10.568	13.22	53.066	24.616	8.258	171.686
128	16	19.628	17.078	17.148	15.812	16.144	8.166	9.998	51.616	27.402	9.42	156.108
256	2	24.166	48.784	10.358	48.706	10.578	12.554	16.54	65.394	55.188	31.484	50.072
256	4	3.626	4.024	19.872	18.242	17.25	8.646	12.608	74.128	3.834	5.746	191.488
256	8	8.126	42.554	18.368	17.408	20.472	8.206	13.686	74.166	15.184	11.504	197.96
256	16	20.284	31.616	21.82	47.022	20.076	7.704	13.1	67.702	18.308	20.596	196.942
512	2	119.51	162.66	13.602	63.524	11.02	14.518	13.34	104.58	125.31	79.73	209.946
512	4	4.326	2.575	43.154	4.368	35.744	9.168	11.394	111.702	3.8075	5.124	251.942
512	8	33.894	8.27	54.972	27.008	47.826	16.624	14.462	111.16	21.002	18.31	255.956
512	16	58.466	75.662	66.006	67.376	61.222	19.282	25.212	114.13	50.3	51.462	276.726
1024	2	80.638	146.723333333	28.904	237.21	27.43	23.238	18.578	189.25	129.27	112.24	363.182
1024	4	22.814	24.1825	69.232	2.425	44.644	17.168	13.572	185.426	49.516	54.3466666667	364.48
1024	8	89.93	131.158	96.976	67.412	82.326	37.786	41.86	200.578	74.014	141.752	394.536
1024	16	124.684	175.168	119.374	142.456	97.264	43.94	47.218	204.856	119.794	111.414	408.01

Figure A.3: LMBench CTX with 4 cores.

Size	Procs	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0	2	20.83	27.756	6.808	20.12	6.736	7.68	9.788	6.5	28.612	29.556	60.588
0	4	63.972	58.474	7.986	31.836	8.49	9.102	11.974	6.336	30.546	24.166	127.562
0	8	28.376	32.164	8.998	28.616	9.146	7.064	9.754	9.866	47.542	31.346	147.016
0	16	18.692	33.62	10.238	53.518	9.56	9.762	10.364	8.998	12.592	12.838	148.996
8	2	42.618	21.678	6.882	31.876	6.774	10.524	9.93	5.952	26.118	28.708	59.95
8	4	71.222	58.604	8.374	50.4	8.2	10.12	11.162	13.878	61.05	49.492	139.442
8	8	33.232	37.88	9.718	25.592	9.646	8.334	11.688	8.188	64.224	26.334	148.238
8	16	59.742	57.646	10.204	30.57	10.578	9.864	14.592	10.866	21.122	4.238	138.112
16	2	43.25	22.658	7.07	40.118	6.882	8.632	10.358	5.39	39.382	27.256	61.542
16	4	65.0	57.946	8.6	82.552	8.562	9.51	11.848	11.282	73.07	46.97	137.95
16	8	37.306	17.168	10.506	8.466	10.154	9.32	13.196	14.62	53.656	19.27	143.916
16	16	31.416	24.494	12.234	35.348	11.59	9.894	14.252	9.58	32.604	28.082	152.038
32	2	24.75	31.2	7.684	30.716	6.564	10.562	9.902	8.29	70.804	20.864	66.404
32	4	65.268	36.522	8.83	52.68	8.786	8.75	11.702	23.602	42.494	19.164	140.704
32	8	20.178	18.71	10.132	39.35	10.38	9.522	14.502	9.47	51.502	39.86	150.74
32	16	44.23	27.206	12.908	53.754	11.994	11.054	15.986	20.42	38.922	31.446	158.246
64	2	56.032	50.434	7.344	58.082	7.54	11.266	10.552	9.348	58.676	32.582	87.524
64	4	51.812	77.238	9.706	48.208	9.03	11.29	13.148	34.288	82.236	59.644	157.18
64	8	27.454	48.446	13.064	34.298	11.518	10.64	16.006	19.988	53.81	32.386	164.042
64	16	32.506	37.9	13.372	57.452	13.45	13.21	17.812	28.26	59.762	19.572	165.33
128	2	64.004	47.906	9.626	58.166	9.344	11.646	12.902	21.36	86.618	58.576	74.618
128	4	88.8	107.206	15.622	74.48	11.432	13.958	15.37	46.22	117.612	36.362	165.18
128	8	43.568	19.576	13.774	46.592	14.688	15.186	20.984	53.066	90.766	21.454	185.98
128	16	88.67	74.536	16.57	87.29	15.518	15.306	21.05	51.616	83.932	25.468	186.78
256	2	74.56	59.434	11.09	35.794	10.572	12.878	12.43	65.394	75.274	98.398	84.226
256	4	133.3	92.818	17.402	29.644	17.686	22.07	21.61	74.128	119.51	70.822	215.9
256	8	101.11	51.722	22.752	52.766	18.09	26.36	27.34	74.166	75.876	63.09	226.092
256	16	92.57	40.618	20.398	45.03	17.954	23.86	28.198	67.702	93.288	40.574	225.696
512	2	156.472	129.338	12.338	95.666	11.776	18.038	14.188	104.58	173.878	82.778	177.86
512	4	142.84	176.546	48.45	134.71	15.164	46.478	31.368	111.702	148.112	77.028	274.01
512	8	95.5	114.892	60.158	155.57	48.474	42.926	38.844	111.16	127.118	38.316	283.626
512	16	100.486	95.12	69.932	118.726	35.582	59.468	49.39	114.13	129.17	90.276	298.782
1024	2	231.51	240.936	23.236	235.146	21.906	25.384	19.96	189.25	239.108	167.89	237.932
1024	4	213.03	70.795	47.868	248.208	40.91	55.11	55.398	185.426	208.214	194.452	380.99
1024	8	68.584	211.568	111.824	215.13	108.916	73.414	80.08	200.578	203.254	86.982	434.292
1024	16	200.914	174.074	128.972	197.456	107.514	92.916	85.086	204.856	120.638	169.566	461.574

Figure A.4: LMBench CTX with 8 cores.

A.3 LMBench MEM

Size	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0.001024	50254.618	52882.902	43866.01	50376.182	43893.814	43812.99	43219.076	51117.75	50103.036	52673.738	47355.14
0.002048	52085.1	53857.404	44851.588	50986.702	44127.404	44805.61	44235.546	53635.67	50494.596	53772.176	48285.558
0.004096	51751.868	54543.934	45417.97	52073.308	45432.534	45365.502	44952.52	54311.026	52172.858	54037.17	48333.382
0.008192	52790.594	54562.052	45719.35	51903.912	44744.056	45667.744	44863.908	55563.262	52250.814	54720.632	48953.498
0.016384	52520.638	54724.036	45934.376	52021.582	45932.628	45784.458	45352.364	54871.13	52627.956	54885.904	49977.174
0.032768	52591.882	54498.432	45695.51	52595.242	42075.832	45648.838	44951.26	53978.078	52053.568	53975.826	49337.55
0.065536	35408.288	36845.16	30958.488	35618.994	30986.718	30947.71	30478.662	36902.02	35123.17	36323.784	33183.694
0.131072	35073.842	36846.506	30944.868	35832.246	30816.854	30795.178	30193.366	35355.128	34369.518	36541.35	32577.418
0.262144	30950.658	32921.516	27397.268	32534.12	27943.878	26711.86	26748.6	31997.004	31015.8	32570.27	29031.92
0.524288	25927.872	27537.924	23162.774	26977.652	23153.082	23154.436	22880.306	27353.162	26177.19	27602.284	24696.246
1.05	26705.166	27411.326	23056.576	26102.872	23063.948	23019.124	22758.606	27722.04	25776.814	27375.652	24490.446
2.10	26086.696	27218.168	23026.49	26119.984	23048.178	22929.256	22559.168	26470.55	25674.7	26734.802	23695.17
4.19	25799.424	27626.262	22577.634	26409.426	22559.81	23103.608	22288.584	26770.098	26463.206	27214.494	23027.836
8.39	18275.682	20845.822	18250.412	19775.57	16426.512	19437.452	18016.728	19385.72	18953.758	20652.718	15874.834
16.78	11737.756	14171.142	14048.99	11826.444	14180.98	14436.736	13595.134	14046.986	12636.238	13808.674	12094.048
33.55	12322.59	13352.828	13881.086	12797.904	13380.582	14472.682	13709.766	13586.884	12276.434	13620.396	12215.996
67.11	13021.556	14548.332	14094.074	12664.93	14121.948	14495.308	13555.998	12594.628	13289.984	13999.902	11852.882
134.22	12291.532	14071.178	13803.728	12042.976	13677.804	14501.254	13578.1	12633.37	12672.344	14405.784	12204.624
268.44	12500.904	14473.972	13745.002	13253.1	13848.732	14511.752	13732.572	13366.924	12420.23	14168.728	12253.522
536.87	11758.754	14567.972	13894.222	12866.576	13926.064	14516.23	13551.198	13168.936	12854.782	14185.924	12506.014
1073.74	12184.646	14343.594	13881.002	12418.322	14104.006	14510.066	13691.372	12922.632	12204.144	13761.066	11898.556

Figure A.5: LMBench MEM read with 1 core data.

Size	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0.001024	50607.308	50158.328	43677.93	51613.128	43147.998	43832.982	43227.102	50515.51	49916.822	51382.98	44131.15
0.002048	50761.994	52456.942	44165.758	53129.998	41445.276	44790.248	44440.236	50962.58	51601.724	51373.086	46721.304
0.004096	51313.568	52714.0	45192.28	53388.71	44624.856	45362.176	44791.746	52595.782	52403.536	52949.152	49185.63
0.008192	52877.188	52853.506	44860.294	53309.142	44730.674	45666.914	44984.606	52607.738	53338.36	52562.972	48898.99
0.016384	52888.138	52594.612	45663.956	53829.838	44800.596	45662.804	45291.2	53151.37	53406.192	52797.452	44377.202
0.032768	52060.304	52586.602	45031.84	53167.898	43685.336	45276.222	44770.696	52440.952	52515.214	53813.628	45013.218
0.065536	34949.676	34764.484	30979.462	36175.91	30487.796	30943.808	30967.348	36306.476	36001.882	35444.606	29527.762
0.131072	34585.536	35786.106	30463.418	35793.608	30227.49	30892.852	30647.132	36347.072	35454.962	35182.218	33222.436
0.262144	30990.51	31822.502	26793.84	32670.846	27361.282	27290.2	27739.694	32077.152	31078.658	32022.78	26344.108
0.524288	26512.92	26969.21	22928.646	26916.374	23013.572	23344.1	23156.138	27405.404	26375.084	26740.57	24741.368
1.05	26009.99	26143.366	23026.344	26963.49	22277.86	23027.462	22823.082	26955.99	26755.986	26499.8	23326.852
2.10	25823.612	25872.258	22685.296	26230.358	22570.328	22924.928	22790.018	26525.098	26427.328	26622.358	23246.602
4.19	26617.946	26143.606	22568.498	27044.546	22167.43	23031.746	22540.81	26949.222	26756.912	26110.53	22437.32
8.39	18640.556	19246.684	18080.04	20810.37	16892.95	19456.066	18058.358	20565.968	19125.028	18022.51	14815.606
16.78	12508.276	11632.554	14012.026	13170.908	13075.358	14440.71	13411.696	14064.364	13127.186	13086.774	10551.324
33.55	12294.264	14219.812	13905.428	14762.348	12772.366	14462.296	13276.934	13293.924	12566.094	12999.126	11078.558
67.11	12621.828	12114.144	14031.75	13105.698	12325.166	14202.388	13153.612	13412.724	12792.216	13013.552	11178.132
134.22	11723.7	13884.176	13978.176	14538.32	13132.392	14486.86	13699.51	14660.074	13566.792	12978.698	10856.0
268.44	12200.92	12260.542	14063.266	13640.26	13051.312	14504.072	13489.696	13232.576	12485.48	12661.036	10672.054
536.87	12719.55	13943.598	14068.844	14082.872	13434.282	14488.108	13608.944	13933.924	12379.284	12593.924	11362.96
1073.74	11662.568	12543.838	14044.39	13960.272	13500.84	14438.452	13533.692	14552.586	12709.662	12367.446	10431.64

Figure A.6: LMBench MEM read with 2 cores data.

Size	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0.001024	50553.46	53362.562	43993.716	51170.232	43875.34	43834.468	43274.276	51188.182	49374.308	52849.392	47964.728
0.002048	51694.394	54296.064	45221.136	52490.142	44770.522	45153.816	44701.028	53062.636	51273.888	53468.634	49461.906
0.004096	52268.144	55134.35	45917.874	53254.356	45704.13	45848.846	45302.818	53028.956	53537.92	54997.204	49537.93
0.008192	53250.268	55493.67	45997.084	53598.102	45391.03	46209.77	45640.338	53942.314	53846.636	55089.168	49260.438
0.016384	53155.272	55809.06	46408.026	53629.254	45694.036	46375.436	45904.458	52718.682	52992.742	55495.542	49494.494
0.032768	50572.026	53620.68	44287.816	50917.69	44300.884	44260.94	43727.84	51450.638	50470.384	53277.752	46748.882
0.065536	31755.06	32898.58	28271.57	31705.18	27792.216	28139.876	27840.888	32852.204	31708.736	33337.498	29510.472
0.131072	32243.366	32679.708	27806.194	31496.676	27780.382	27765.198	27385.588	31994.274	31243.766	32709.136	28944.73
0.262144	29144.77	30760.596	26721.048	30225.546	26541.658	26286.226	25539.962	31032.51	28440.934	30553.574	25710.296
0.524288	18243.962	19947.204	19145.276	19040.01	19253.972	19401.262	18783.98	20441.192	18693.272	20727.282	17341.712
1.05	18281.676	19479.294	18333.182	18373.748	17865.504	18466.378	17977.87	18932.97	18104.48	20055.262	16748.77
2.10	18246.94	20155.652	18474.122	18096.908	18483.44	18413.21	18136.81	17879.938	18818.418	20105.382	16664.16
4.19	17569.228	20187.574	18402.13	18103.36	17698.686	18330.172	17875.402	18317.168	18555.662	19871.102	16328.168
8.39	12746.456	13422.694	11942.784	12626.192	11156.666	13455.19	12172.71	13555.802	12808.042	13361.754	10055.964
16.78	6431.942	6971.782	7106.376	6759.328	7131.202	6844.82	6913.126	7114.194	6488.43	6940.236	5968.87
33.55	6815.938	6875.554	7038.518	6558.326	6876.846	6796.822	6921.03	6810.144	6434.312	6975.248	5844.706
67.11	6607.4	6889.66	7011.31	6701.446	6668.498	6831.824	6922.17	6652.094	6525.542	7111.43	6009.23
134.22	6678.114	6889.12	7032.406	6684.526	7029.54	6852.95	6962.384	6734.516	6569.362	7076.374	5903.28
268.44	6640.196	6901.842	6987.81	6643.62	6956.18	6864.09	6949.316	6819.144	6561.624	7086.114	6001.24
536.87	6651.736	6890.634	6932.118	6666.998	6919.7	6870.242	6942.078	6711.706	6620.818	7094.052	5935.244
1073.74	6628.398	6880.72	6915.84	6743.942	6892.686	6872.484	6956.63	6666.822	6580.014	7052.29	5905.554

Figure A.7: LMBench MEM write with 1 core data.

Size	qemu-kvm-host-virtio	qemu-kvm	xen-pv	qemu-1.2.2-ekvm	libvirt-xen-pv	xen-hvm	libvirt-xen-hvm	host	qemu-kvm-host	libvirt-kvm	virtualbox
0.001024	50224.304	50947.91	43804.774	52571.152	42806.164	43831.232	43195.286	51840.468	50184.456	50591.594	48195.594
0.002048	50831.956	52989.748	44714.52	54124.302	44458.83	45139.238	44386.054	52119.57	53237.882	52958.116	44410.928
0.004096	51972.628	54359.688	45841.084	54206.108	44872.738	45850.194	45779.146	54085.616	52609.592	52469.24	49134.404
0.008192	51877.252	52863.79	46014.868	54425.832	44487.884	46203.68	45802.71	55131.612	53790.58	53117.738	47831.476
0.016384	53306.128	52573.398	46185.39	55573.5	45249.042	46358.912	46105.392	54663.548	52800.208	53580.95	49364.974
0.032768	50895.498	50126.996	43819.834	51757.022	43270.794	44219.696	43269.19	52151.276	50449.818	52352.384	47358.846
0.065536	31698.468	31446.336	27948.612	32702.22	27655.26	27981.43	27550.202	33605.33	31910.552	30823.25	30274.06
0.131072	31042.16	31388.566	27441.4	32153.916	26958.388	27810.798	27742.484	32988.002	31772.144	31550.02	28509.618
0.262144	30255.06	29321.394	26550.79	29802.896	26262.464	26153.082	26437.828	30952.284	28645.38	28742.57	27594.792
0.524288	17081.314	18955.566	19010.478	19752.09	18698.47	19320.436	19127.584	18461.78	19393.404	19009.306	15781.458
1.05	17992.078	18606.942	18351.822	19502.858	18181.776	18461.31	18314.014	18321.482	19014.11	17954.066	16726.198
2.10	17907.878	18725.622	18176.446	19522.998	18168.904	18408.594	18107.602	19294.414	17830.004	18115.806	14578.9
4.19	17797.518	18824.332	18220.306	19946.904	17884.924	18436.002	18200.154	19026.054	18067.296	17805.068	15912.982
8.39	12201.574	13118.136	11736.084	12889.134	11405.48	13223.648	12463.056	13568.49	13069.404	11958.354	9749.648
16.78	6611.556	6551.626	7111.534	6766.414	7028.706	7066.494	6765.658	7067.516	6450.032	6503.538	6141.0
33.55	6497.542	6549.368	6926.512	6714.602	6865.532	7000.876	6932.686	6762.898	6416.608	6388.008	5985.092
67.11	6640.024	6475.654	6901.652	6775.684	6885.984	6908.194	6984.6	6998.188	6615.456	6466.136	5803.112
134.22	6651.984	6691.074	6886.15	6778.87	6997.288	6905.414	6973.514	6738.586	6705.54	6658.654	5948.198
268.44	6556.43	6602.65	7008.614	6772.394	6861.74	6890.104	6988.408	6797.406	6672.196	6674.458	5903.246
536.87	6594.018	6522.102	6933.996	6765.996	6840.16	6887.626	6996.316	6647.182	6649.394	6600.646	5977.496
1073.74	6658.932	6600.346	6881.154	6827.052	6841.054	6884.44	6984.044	6789.568	6642.61	6619.346	6009.024

Figure A.8: LMBench MEM write with 2 cores data.

Appendix B

Installation of used software

B.1 Introduction

This appendix will give further explanation of how each of the virtualization solutions that have been used in this thesis can be installed on a Linux/UNIX system.

For all the instructions it is assumed that the Fedora *Development Tools* and *Development Libraries* are installed. This is easily done by issuing the following: *yum groupinstall 'Development Tools' 'Development Libraries'*

B.2 KVM

The very first thing to do is to check if your processor is capable of supporting KVM virtual machines. If your machine is newer than 2006, you are most likely to have either Intel VT-x or AMD SVM virtualization support on your processor. To check this is simply done by issuing one of the following commands.

- `grep -E "(vmx|svm)" /proc/cpuinfo`
- `lscpu | grep "Virtualization"`

This should highlight either Intel VT-x called `vmx` or AMD SVM called `svm`. If nothing is returned or highlighted in your terminal, your processor is not going to support KVM virtualization.

In the directory of the `kvm-kmod` source, first issue `./configure` then issue `make` to build the module. Lastly install the module by issuing either `su -c 'make install'` or `sudo make install`.

To insert the module into your kernel use the `modprobe` command to insert the KVM module. Depending on your processor architecture you have either an Intel or AMD processor, which of the kernel module you should use should be apparent. Example below:

- If you have `sudo`: `sudo modprobe kvm_(intel or amd)`
- If you use `su`: `su -c "modprobe kvm_(intel or amd)`

B.3 QEMU

For the version of Qemu used in this thesis I have used version 1.2.2 built from source, the instructions follows.

First unpack the source `tar` file and move into the folder. In the root source directory, *qemu-1.2.2/*, issue `./configure --target-list='x86_64-softmmu' --enable-kvm --disable-tcg`. When the configuration finishes issue `make` to build the Qemu executable, when the build is done install it by issuing `su -c 'make install'` of `sudo make install`.

B.4 QEMU-KVM

To install the Qemu-KVM version used in the benchmarks, which was version 1.2.0, first unpack the source file and change directory to the root source folder.

In this folder, *qemu-kvm-1.2.0/*, issue `./configure` to start the configuration. Then build the executable by issuing `make`, and to install issue either `su -c 'make install'` or `sudo make install`.

B.5 High Performance Linpack (HPL)

Here I will shortly present the instructions for installing HPL on a Fedora distribution. To be able to build some prerequisites are needed to build HPL, these are; *openmpi, openmpi-devel, atlas, atlas-devel, tbb, tbb-devel, compat-gcc-34-g77*. All of which can be installed from the Fedora package manager *yum*.

Then extract the `hpl-2.1.tar.gz` file and go into the directory. The Makefile configuration can be confusing, however the user need only change a few lines. What I have done is copy the `Make.Linux_PII_CBLAS` in the `setup` directory to the root folder of HPL and renamed it to *Make.Linux_CBLAS* and then changed the following lines:

```
64: ARCH  = Linux_CBLAS
70: TOPdir  = /my/current/directory/hpl-2.1
71: INCdir  = /my/current/directory/hpl-2.1/include
84: MPdir  = /usr/lib64/openmpi
85: MPinc = -I/usr/include/openmpi-x86_64
86: MPlib  = /usr/lib64/openmpi/lib/libmpi.so.1
95: LAdir  = /usr/lib64/atlas
96: LAinc = -I/usr/include/atlas
97: LAlib = /usr/lib64/atlas/libcblas.so /usr/lib64/atlas/libatlas.so
```

From there on, in the root directory of HPL, issue `make arch=Linux_CBLAS`. Now the HPL benchmark will build.

The user then need to add the following to the `bashrc` file to be able to run the benchmark.

```
export PATH=$PATH:/usr/lib64/openmpi/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib
```

From the root folder of HPL move to the directory `bin/Linux_CBLAS`, here you should find an executable called `xhpl` and a file called HPL.dat. The HPL.dat file can be tweaked to perform the benchmark with your parameters example file follows. To run the benchmark with 4 processes issue `mpirun -np 4 ./xhpl`, which will run the benchmark with the configuration in the present HPL.dat file.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
8            device out (6=stdout,7=stderr,file)
8            # of problems sizes (N)
1000 3000 5000 7000 9000 11000 13000 15000      Ns
1            # of NBs
82           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
2            Ps
4            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

Figure B.1: HPL configuration file.

Appendix C

Virtualization suite configuration

This chapter will present the commands, parameters and the configuration files used for Qemu/KVM and Xen for the benchmarking in this thesis. The various disk configurations is not included, as they are supplied as disk images and there is no need for format specification. Except in the case of LVM where this is given as an example for Qemu-kvm and Xen-HVM-LVM.

C.1 Qemu/KVM

Qemu-1.2.2

```
qemu-system-x86_64 -m 2048 -k no -smp # -hda disk.img
        --enable-kvm -nographic --redir tcp:2222::22
```

Qemu-kvm

```
qemu-system-x86_64 -m 2048 -k no -smp # -hda disk.img
        -nographic --redir tcp:2222::22
```

Qemu-kvm, with LVM

```
qemu-system-x86_64 -m 2048 -k no -smp #
        -hda /dev/vg_virt/kvm\_virt -nographic
        --redir tcp:2222::22
```

Qemu-kvm-host

```
qemu-system-x86_64 -m 2048 -k no -smp # -hda disk.img
        -nographic --redir tcp:2222::22 --cpu host
```

Qemu-kvm-host-virtio

```
qemu-system-x86_64 -m 2048 -k no -smp # -nographic
        --redir tcp:2222::22 -drive file=disk.img,if=virtio
        --cpu host
```

C.2 Xen

Xen-HVM

```
builder = 'hvm'
memory = 2048
vcpus = #
name = "Fedora-HVM"
vif = ['bridge=virbr0']
disk = ['tap:aio:var/lib/xen/images/
        xl-fedora-xen-hvm,xvda,w']
boot = 'cd'
keymap = 'no'
```

Xen-HVM-LVM

With regular LVM configuration, as well as with the aforementioned optimizations.

```
builder = 'hvm'
memory = 2048
vcpus = #
name = "Fedora-HVM-LVM"
vif = ['bridge=virbr0']
disk = ['phy:/dev/vg_virt/virt_storage,xvda,w']
#disk = ['phy:/dev/vg_virt/virt_storage,hda,w']
xen_platform_pci=1
boot = 'cd'
keymap = 'no'
```

Xen-PV

```
name = "Fedora-PV"
builder = "generic"
vcpus = #
memory = 2048
vif = ['bridge=virbr0']
disk = ["tap:aio:/var/lib/xen/images/
        xl-fedora-xen-pv.img,xvda,w"]
bootloader = "pygrub"
#kernel = 'vmlinuz'
#ramdisk = 'initrd.img'
#extra = 'root=live:http://dl.fedoraproject.org/pub/ \
        fedora/linux/releases/17/Fedora/x86_64/os/ \
        LiveOS/squashfs.img inst.headless' \
keymap = 'no'
on_reboot = "restart"
on_crash = "restart"
```

Bibliography

[1] R.J. Adair, R.U. Bayles, L.W. Comeau, and R.J. Creasy. A virtual machine system for the 360/40. Technical report, IBM Cambridge Scientific Center report 320-2007, May 1966.

[2] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4):3–18, December 2010.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen 2002. Technical Report 553, University of Cambridge, January 2003.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[5] Nikhil Bathia. Performance evaluation of intel ept hardware assist. Technical report, VMWare, 2008.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[7] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 574–579. European Design and Automation Association, 2010.

[8] Jr. William I. Bullers, Stephen Burd, and Alessandro F. Seazzu. Virtual machines - an idea whose time has returned: application to network, security, and database courses. *SIGCSE Bull.*, 38:102–106, March 2006.

[9] Damien Cerbelaud, Shishir Garg, and Jeremy Huylebroeck. Opening the clouds: qualitative overview of the state-of-the-art open source vm-based cloud management platforms. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, page 22. Springer-Verlag New York, Inc., 2009.

[10] Jianhua Che, Qinming He, Qinghua Gao, and Dawei Huang. Performance measuring and comparing of virtual machine monitors. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, volume 2, pages 381 –386, dec. 2008.

[11] Jianhua Che, Yong Yu, Congcong Shi, and Weimin Lin. A synthetical performance evaluation of openvz, xen and kvm. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 587 –594, dec. 2010.

[12] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483 –490, sep. 1981.

[13] Todd Deshane, Zachary Shepherd, J Matthews, Muli Ben-Yehuda, Amit Shah, and Balaji Rao. Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA*, pages 1–2, 2008.

[14] P. Domingues, F. Araujo, and L. Silva. Evaluating the performance and intrusiveness of virtual machines for desktop grid computing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009.

[15] Michael Fenn, Michael A Murphy, and Sebastien Goasguen. A study of a kvm-based cluster for grid computing. In *Proceedings of the 47th Annual Southeast Regional Conference*, page 34. ACM, 2009.

[16] John Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Commun. ACM*, 4:435–436, October 1961.

[17] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[18] James Gray. Readers choice awards 2010, 13. February 2010. http://www.linuxjournal.com/content/readers-choice-awards-2010.

[19] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[20] Stefan Hajnoczi. Kvm architecture overview. Presentation.

[21] Stefan Hajnozci. Should i use qemu or kvm, October, 22. 2012. http://blog.vmsplice.net/2011/03/should-i-use-qemu-or-kvm.html.

[22] Joab Jackson. Red hat drops xen from rhel, May 2012. http://www.computerworld.com/s/article/9175883/Red_Hat_drops_Xen_from_RHEL.

[23] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[24] Vipin Kumar. Hpl application note, March 2013. http://software.intel.com/en-us/articles/performance-tools-for-software-developers-hpl-application-note.

[25] Oren Laadan and Jason Nieh. Operating system virtualization: practice and experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 17:1–17:12, New York, NY, USA, 2010. ACM.

[26] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[27] Meriam Mahjoub, Afef Mdhaffar, Riadh Ben Halima, and Mohamed Jmaiel. A comparative study of the current cloud computing technologies and offers. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 131–134. IEEE, 2011.

[28] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199 –218, 1970.

[29] Karissa Miller and Mahmoud Pegah. Virtualization: virtually at the desktop. In *Proceedings of the 35th annual ACM SIGUCCS fall conference*, SIGUCCS '07, pages 255–260, New York, NY, USA, 2007. ACM.

[30] Mozilla. Mozilla firefox - portable edition, March 2012. http://portableapps.com/apps/internet/firefox_portable.

[31] Oracle. *VirtualBox User manual, Chpater 10. Technical Background.*

[32] Oracle. *VirtualBox User manual, Chpater 6. Virtual Networking.*

[33] A. Padegs. System/360 and beyond. *IBM J. Res. Dev.*, 25(5):377–390, September 1981.

[34] Oracle White Paper. Best practices for running oracle databases in oracle solaris containers. Technical report, Oracle, June 2011.

[35] PhysTech. Ubench benchmark, April 2013. http://www.phystech.com/download/ubench.html.

[36] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974.

[37] Proceedings of SEAS AM82. *CP-40, the Origin of VM/370*, May 1982.

[38] Emerson Pugh, Lyle R. Johnson, and John H. Palmer. *IBM's 360 and early 370 systems.* MIT Press, Cambridge, MA, USA, 1991.

[39] Nathan Regola and Jean-Christophe Ducom. Recommendations for virtualization technologies in high performance computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 409–416, Washington, DC, USA, 2010. IEEE Computer Society.

[40] A. Ribiere. Emulation of obsolete hardware in open source virtualization software. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 354–360, 2010.

[41] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, SSYM'00, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[42] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2:34–40, July 2004.

[43] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[44] Kristoffer Robin Stokke. Gpu virtualization. 2012.

[45] Christopher Strachey. Time sharing in large, fast computers. In *IFIP Congress*, pages 336–341, 1959.

[46] Honglin Su. Oracle vm blog: Basics of oracle vm, May 2012. https://blogs.oracle.com/virtualization/entry/oracle_vm_blog_basics_of_oracl.

[47] I. Tafa, E. Beqiri, H. Paci, E. Kajo, and A. Xhuvani. The evaluation of transfer time, cpu consumption and memory utilization in xen-pv, xen-hvm, openvz, kvm-fv and kvm-pv hypervisors using ftp and http approaches. In *Intelligent Networking and Collaborative Systems (INCoS), 2011 Third International Conference on*, pages 502–507, 2011.

[48] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[49] Texmaker. Texmaker - latex editor, March 2012. http://www.xm1math.net/texmaker/.

[50] G.K. Thiruvathukal, K. Hinsen, K. Laandufer, and J. Kaylor. Virtualization for computational scientists. *Computing in Science Engineering*, 12(4):52–61, july-aug. 2010.

[51] Melinda Varian. Vm and the vm community: Past, present, and future. office of computing and information technology. Technical report, Princeton University, Princeton, NJ, 1997.

[52] VirtualBox. Developer faq, April 2013. https://www.virtualbox.org/wiki/Developer_FAQ.

[53] David Walden and Tom Van Vleck. *The Compatible Time Sharing System (1961 - 1973) Fiftieth Anniversary Commemorative Overview*. IEEE Computer Society, 2011.

[54] Aaron Weiss. Computing in the clouds. *netWorker*, 11:16–25, December 2007.

[55] KVM wiki. Kvm guest support status, April 2013. http://www.linux-kvm.org/page/Guest_Support_Status.

[56] Qemu wiki. Kvm, April 2013. http://wiki.qemu.org/KVM.

[57] VirtualBox wiki. Guest oses, April 2013. https://www.virtualbox.org/wiki/Guest_OSes.

[58] Xen wiki. Credit scheduler, April 2013. http://wiki.xen.org/wiki/Credit_Scheduler.

[59] Xen wiki. Dom0 kernels for xen, April 2013. http://wiki.xen.org/wiki/Dom0_Kernels_for_Xen.

[60] Xen wiki. Domu support for xen, April 2013. http://wiki.xen.org/wiki/DomU_Support_for_Xen.

[61] Xen wiki. Hvmloader, March 2013. http://wiki.xen.org/wiki/Hvmloader.

[62] Xen wiki. Xen linux pv on hvm drivers, March 2013. http://wiki.xen.org/wiki/Xen_Linux_PV_on_HVM_drivers.

[63] Xen wiki. Xl, March 2013. http://wiki.xen.org/wiki/XL.

[64] Wikipedia, March 2012. http://en.wikipedia.org/wiki/IBM_CP_40.

[65] Wikipedia, March 2012. http://en.wikipedia.org/wiki/History_of_CP/CMS.

[66] Wikipedia. Hypervisor, 3. February 2012. http://en.wikipedia.org/wiki/Hypervisor.

[67] Xianghua Xu, Feng Zhou, Jian Wan, and Yucheng Jiang. Quantifying performance properties of virtual machine. In *Information Science and Engineering, 2008. ISISE '08. International Symposium on*, volume 1, pages 24 –28, dec. 2008.

[68] Andrew J Younge, Robert Henschel, James T Brown, Gregor von Laszewski, Judy Qiu, and Geoffrey C Fox. Analysis of virtualization technologies for high performance computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 9–16. IEEE, 2011.